NPS ARCHIVE
1999.03
LINK, J.

NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

DESIGN OF A SERIAL COMMUNICATION PROTOCOL AND BUS INTERFACE CHIP FOR TACTILE COMMUNICATIONS

by

Jeffrey P. Link

March 1999

Thesis Advisors: Douglas J. Fouts
Jon T. Butler

THESIS
L66346

Approved for public release; distribution is unlimited.

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5101

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188.

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1999	Master's Thesis

4. TITLE AND SUBTITLE
DESIGN OF A SERIAL COMMUNICATION PROTOCOL AND BUS INTERFACE CHIP FOR TACTILE COMMUNICATIONS

5. FUNDING NUMBERS

6. AUTHOR(S)
Jeffrey P. Link

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Tactile communication requires rapid data transfer along a common bus. The developed communication protocol and application-specific interface chip enable precise control of multiple tactile transmitters (tactors) to convey information to military users. This extrapolation of the Tactile Situation Awareness System developed by the Naval Aerospace Medical Research Laboratory uses a serial data bus and individual interface chips to communicate commands with a minimum number of conductors. This thesis develops the communication protocol and the design of the Tactor Interface Chip (TIC). This work also includes a computer-driven tactile array controller and Parallel Port Data Modulator for TIC testing and demonstration.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Electronics, Tactile, Interface, Tactor, TSAS, Serial Communications	316
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18, 298-102

Approved for public release; distribution is unlimited.

DESIGN OF A SERIAL COMMUNICATION PROTOCOL AND BUS INTERFACE CHIP FOR TACTILE COMMUNICATIONS

Jeffrey P. Link
Lieutenant Commander, United States Navy
B.S., Iowa State University, 1985
M.A., Webster University, 1991

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
March 1999

ABSTRACT

Tactile communication requires rapid data transfer along a common bus. The developed communication protocol and application-specific interface chip enable precise control of multiple tactile transmitters (tactors) to convey information to military users. This extrapolation of the Tactile Situation Awareness System developed by the Naval Aerospace Medical Research Laboratory uses a serial data bus and individual interface chips to communicate commands with a minimum number of conductors. This thesis develops the communication protocol and the design of the Tactor Interface Chip (TIC). This work also includes a computer-driven tactile array controller and Parallel Port Data Modulator for TIC testing and demonstration.

TABLE OF CONTENTS

I. INTRODUCTION 1

A. COMMUNICATING THROUGH TOUCH 2

B. TACTILE SITUATION AWARENESS SYSTEM BY NAMRL 3

C. DEVELOPING INTELLIGENT TACTORS 4

D. THESIS OUTLINE 5

II. COMMUNICATION PROTOCOL 7

A. DESIGN REQUIREMENTS 7

1. Required Output 7

2. System Configuration 8

B. CONTROL STRUCTURE 9

1. Address 9

2. Pulse Shape 9

C. COMMAND FORMAT 10

1. Address Command Word 10

2. Pulse Width Word 11

3. Repetition Period Word 12

D. BUS ARCHITECTURAL CONSIDERATIONS 12

1. Command Promulgation Speed 13

2. Parallel Bus 13

3. Serial Bus 14

E. ARCHITECTURAL DECISION AND JUSTIFICATION 15

F. TRANSMISSION PACKET FORMAT 15

G. PHYSICAL CONSTRUCTION REQUIREMENTS 16

1. TIC and Tactor Power 17

2. Command Data and Timing Signals 17

III. TACTOR INTERFACE CHIP SPECIFICATION AND DESIGN 19
A. DESIGN GOALS 19
B. OPERATIONAL CONCEPT 20
1. Serial Data Receiver 21
2. Command Decoder and Controller 21
3. Tactor Power Controller 21
C. OPERATIONAL DESCRIPTION 22
D. FUNCTIONAL MODULE DESIGN 23
1. Serial Data Receiver 23
2. Command Decoder and Controller 24
3. Tactor Power Controller 26
E. VERILOG® DESIGN VERIFICATION 27
1. Twelve-Bit Input Shift Register 28
2. Eight-Bit Data Latch 28
3. Input Stream Validity Check 29
4. Command Sequence Controller 31
5. Address Comparator 34
6. Address Reference 34
7. Pulse Width Register 35
8. Repetition Period Register 36
9. Power Control Logic 37
10. Power Oscillator 38
11. Pulse Width Down Counter 39
12. Repetition Period Down Counter 40
13. Clock Divider 41
F. STRUCTURAL COMPONENT DESIGN 42
1. Structural Circuit Optimization 43
2. Twelve-Bit Input Shift Register 44
3. Eight-Bit Data Latch 44

4. Input Stream Validity Check 44
5. Command Sequence Controller 45
6. Address Comparator 46
7. Address Reference 46
8. Pulse Width Register 46
9. Repetition Period Register 47
10. Power Control Logic 47
11. Power Oscillator 47
12. Pulse Width Down Counter 47
13. Repetition Period Down Counter 48
14. Clock Divider 49
G. ADVANCED DESIGN FEATURES 49
1. Multiple Command Packet Addressing 49
2. All-Call Address 50
3. Dual Reset Circuit 50
4. Selectable Oscillator Frequency 50
5. Selectable Address 50
H. ANIMATION OF TACTOR INTERFACE CHIP OPERATIONS 51
1. TIC Visual Representation 51
2. Animation Color Scheme 52
IV. TACTOR INTERFACE CHIP VLSI IMPLEMENTATION 55
A. COMPETING VLSI DESIGN CONSTRAINTS 55
1. Size 55
2. Power 55
3. Speed 56
B. CMOS FET TRANSISTOR SIZING 56
1. Determining PFET Size From NFET Size 56
2. Basic Response Timing 57
B. LOGIC ELEMENT DESIGN 59

C. COMBINED LOGIC COMPONENT CONSTRUCTION 59
E. MODULE ASSEMBLY 60
F. INPUT AND OUTPUT CONSIDERATIONS 60
G. COMPLETE TACTOR INTERFACE CHIP 61
H. COMPREHENSIVE SYSTEM TESTING 63

V. PARALLEL PORT DATA MODULATOR 65
A. CREATING A SERIAL COMMAND STREAM 65
B. PARALLEL PORT INTERFACING AND CONTROL 66
C. MODULATOR DESIGN SPECIFICS 68
D. CREATING A PRINTED CIRCUIT BOARD LAYOUT 69
E. PRINTED CIRCUIT BOARD MANUFACTURING 72
F. PRINTED CIRCUIT BOARD ASSEMBLY 73
G. SOFTWARE TO DRIVE THE COMMAND MODULATOR 74
H. COMMAND MODULATOR TESTING 75
I. MODIFICATIONS TO THE MODULATOR DESIGN 76

VI. TACTOR INTERFACE CHIP TESTING 77
A. VLSI CHIP RECEIPT FROM FABRICATION 77
B. VISUAL INSPECTION 78
C. OPERATIONAL CHECK USING COMMAND MODULATOR 78
D. COMPLETE SYSTEM RESIMULATION 79
E. SCANNING ELECTRON MICROSCOPE INSPECTION 79
F. CHARGED ELECTRON IMAGING 81
G. FURTHER TESTING 82

VII. REVISIONS TO THE COMMUNICATION PROTOCOL 83
A. EVALUATION OF REGISTER COMMAND PAIRS 83
B. TACTILE ARRAY SIZING 84
C. PROGRAMMABLE OSCILLATION FREQUENCY 84
D. COMMANDED RESET 85
E. REVISED COMMANDED STRUCTURE 85

VIII. INCORPORATION OF ADDITIONAL DESIGN FEATURES 87
A. IMPROVED BI-DIRECTIONAL CURRENT SWITCHING SCHEME 87
B. PROGRAMMABLE OSCILLATION FREQUENCY 89
C. WAVE SHAPE GENERATION USING DUTY CYCLE 90
D. REVISED COMMAND DECODER AND CONTROLLER 91

IX. CONCLUSIONS AND FURTHER WORK 93
A. TACTILE INTERFACE SYSTEM PERFORMANCE 93
1. Simulation Performance during Design Process 93
2. Parallel Port Data Modulator Performance 93
3. Manufactured TIC Performance 94
B. IMPROVEMENTS THAT ARE READY TO INCORPORATE 94
1. Expanded Communication Protocol 94
2. Shaped Oscillation Current 94
3. Programmable Frequency 95
C. RECOMMENDATIONS FOR NEXT VLSI LAYOUT 95
1. Elaborate Testing and Measurement Points 95
2. Timing with Up Counters and Comparators 95
D. PROSPECTS FOR FUTURE DEVELOPMENT 95
1. On-board Current Switching 95

2. Programmable Addressing 96
3. Two-Way Communications 96

APPENDIX A. TIC MODELING USING VERILOG 97
A. TACTOR INTERFACE CHIP 97
B. SERIAL DATA RECEIVER 100
1. Twelve-Bit Input Shift Register 102
2. Eight-Bit Data Latch 105
3. Input Stream Validity Check 107
C. COMMAND DECODER AND CONTROLLER 110
1. Command Sequence Controller 113
2. Address Comparator 117
3. Address Reference 120
4. Pulse Width Register 121
5. Repetition Period Register 124
D. TACTOR POWER CONTROLLER 127
1. Power Control Logic 129
2. Power Oscillator 131
3. Pulse Width Down Counter 133
4. Repetition Period Down Counter 136
5. Clock Divider 139
E. SUPPORT COMPONENTS 142
1. Clock with Parametric Half-Period 142
2. D flip-flop, positive edge triggered 143
3. Transmission Gate MUX 145

APPENDIX B. SYSTEM DESIGN SCHEMATICS 147
A. SERIAL DATA RECEIVER 147
1. Twelve-Bit Input Shift Register 147

2. Eight-Bit Data Latch 148
3. Input Stream Validity Check 148
C. COMMAND DECODER AND CONTROLLER 149
1. Command Sequence Controller 149
2. Address Comparator 150
3. Pulse Width Register 151
4. Repetition Period Register 152
D. TACTOR POWER CONTROLLER 153
1. Power Control Logic 153
2. Power Oscillator 153
3. Pulse Width Down Counter 154
4. Repetition Period Down Counter 155
5. Clock Divider 156

APPENDIX C. STRUCTURAL EVALUATION USING SPICE 157
A. GENERAL DEFINITION FILES 157
1. CMOS FET Model Parameters 158
2. Fundamental Logic Element Definitions 159
B. SERIAL DATA RECEIVER 164
1. Twelve-Bit Input Shift Register 167
2. Eight-Bit Data Latch 169
3. Input Stream Validity Check 172
C. COMMAND DECODER AND CONTROLLER 176
1. Command Sequence Controller 182
2. Address Comparator 185
3. Pulse Width Register 188
4. Repetition Period Register 191
D. TACTOR POWER CONTROLLER 194
1. Power Control Logic 198

2. Power Oscillator 198
3. Pulse Width Down Counter 200
4. Repetition Period Down Counter 203
5. Clock Divider 206

APPENDIX D. TACTILE INTERFACE ANIMATION PROGRAM 209
A. ANIMATION DESIGN 209
1. TIC Visual Representation 209
2. Animation Color Scheme 210
B. ANIMATION PROGRAMMING 211
1. Intelligent Tactor 212
2. Command Byte 216
3. TIC Demo 218
4. Demonstration Applet HTML File 221

APPENDIX E. VLSI LOGIC ELEMENT DESIGN 223
A. LOGIC ELEMENT SCHEMATICS 223
1. Inverter 223
2. Two Input NAND 223
3. Three Input NAND 224
4. Four Input NAND 224
5. Three Input AND 224
6. Four Input AND 225
7. Two Input NOR 225
8. Three Input NOR 225
9. Two Input XOR 226
10. Two Input XNOR 226
11. D Flip Flop with Clear 227
12. Two Input Multiplexer 227
B. LOGIC ELEMENT SPICE SIMULATIONS 227

1. Inverter 228
2. Two Input NAND 228
3. Three Input NAND 229
4. Four Input NAND 230
5. Three Input AND 231
6. Four Input AND 232
7. Two Input NOR 233
8. Three Input NOR 234
9. Two Input XOR 235
10. Two Input XNOR 236
11. D Flip Flop with Clear 237
12. Two Input Multiplexer 238

C. VLSI LAYOUT 239

1. Legend of Layout Layers 239
2. Inverter 240
3. Two Input NAND 241
4. Three Input NAND 242
5. Four Input NAND 243
6. Three Input AND 244
7. Four Input AND 245
8. Two Input NOR 246
9. Three Input NOR 247
10. Two Input XOR 248
11. Two Input XNOR 249
12. D Flip Flop with Clear 250
13. Two Input Multiplexer 251

APPENDIX F. PARALLEL DATA MODULATOR DESIGN 253

A. COMMAND MODULATOR DESIGN USING VERILOG® 253

1. Four-Bit Register with Equality and Parity Calculation 255

2. Control State Machine 256
3. Eight-to-One Multiplexer 259
B. COMMAND MODULATOR IMPLEMENTATION USING ABEL™ 261
1. Four-Bit Register with Equality and Parity Calculation 261
2. Control State Machine 263
3. Eight-to-One Multiplexer 265
C. COMMAND MODULATOR ELEMENT INFORMATION 267
1. Four-Bit Register with Equality and Parity Calculation 268
2. Control State Machine 271
3. Eight-to-One Multiplexer 274

APPENDIX G. COMMAND TRANSMISSION PROGRAM 277
A. PARALLEL PORT COMMAND TRANSMISSION 277

APPENDIX H. GOMAC CONFERENCE PAPER 279

LIST OF REFERENCES 285

INITIAL DISTRIBUTION LIST 287

LIST OF FIGURES

Figure 1. TSAS Tactor Control. 3
Figure 2. Local Tactor Control. 4
Figure 3. Composition of the Intelligent Tactor. 5
Figure 4. Tactor Current-Switching Structure. 8
Figure 5. Tactor Activation Parameters. 9
Figure 6. Standard USART data packet format. 16
Figure 7. TIC Functional Modules. 20
Figure 8. TIC Operating States and Transitions. 22
Figure 9. Serial Data Receiver Elements. 24
Figure 10. Partial clearing prevents Erroneous Command Detection. 24
Figure 11. Command Decoder and Controller Elements. 26
Figure 12. Tactor Power Controller Elements. 27
Figure 13. Alternate Structures for Realizing the Address Comparator. 43
Figure 14. Tactile Interface Animation Basics. 52
Figure 15. Tactile Interface Animation in Progress. 53
Figure 16. Inverter Response for Various PFET Widths. 57
Figure 17. Delay Circuit for Measuring Inverter Response. 58
Figure 18. Inverter Transmission Response for Delay Circuit. 58
Figure 19. Completed Tactor Interface Chip VLSI Design. 62
Figure 20. Layout Map of the Tactor Interface Chip VLSI Design. 63
Figure 21. Simulation Results from the Complete TIC Design. 64
Figure 22. Command Modulator Conceptual Design. 66
Figure 23. Parallel Port Connector with Pins Numbered. 68
Figure 24. Parallel Port Modulator Component Layout. 70
Figure 25. Parallel Port Modulator Top Layer Routing. 71
Figure 26. Parallel Port Modulator Bottom Layer Routing. 71
Figure 27. Command Modulator Top Layer after Machining. 73
Figure 28. Fully Assembled Command Modulator. 74
Figure 29. Command Modulator Output for 19 Command. 75

Figure 30. Command Modulator Output for 218 Command....76
Figure 31. Tactor Interface Chip Pin Assignments....78
Figure 32. Scanning Electron Microscope Images of Potential Shorts....80
Figure 33. Scanning Electron Microscope Images of Aluminum Oxidation....80
Figure 34. Scanning Electron Microscope Image of Mask Failure....81
Figure 35. Scanning Electron Microscope Images of Embedded Impurities....81
Figure 36. Tactor Current-Switching Structure....88
Figure 37. Initial Current Switching Pattern....88
Figure 38. Revised Current Switching Pattern....88
Figure 39. Generating the Oscillation Frequency with Revised Switching....89
Figure 40. Wave Shape Generation using a Duty Cycle Register....90
Figure 41. Revised Command Decoder and Controller module....91
Figure 42. TIC Test Bench Verilog® source code....97
Figure 43. TIC Behavioral model Verilog® source code....99
Figure 44. TIC Structural model Verilog® source code....99
Figure 45. Serial Data Receiver Test Bench Verilog® source code....100
Figure 46. Serial Data Receiver Behavioral model Verilog® source code....101
Figure 47. Serial Data Receiver Structural model Verilog® source code....101
Figure 48. Twelve-Bit Input Shift Register Test Bench Verilog® source code....102
Figure 49. Twelve-Bit Input Shift Register Behavioral model Verilog® source code....103
Figure 50. Twelve-Bit Input Shift Register Structural model Verilog® source code....104
Figure 51. Eight-Bit Data Latch Test Bench Verilog® source code....105
Figure 52. Eight-Bit Data Latch Behavioral model Verilog® source code....106
Figure 53. Eight-Bit Data Latch Structural model Verilog® source code....106
Figure 54. Input Stream Validity Check Test Bench Verilog® source code....107
Figure 55. Input Stream Validity Check Behavioral model Verilog® source code....108
Figure 56. Input Stream Validity Check Structural model Verilog® source code....109
Figure 57. Command Decoder and Controller Test Bench Verilog® source code....110
Figure 58. Command Decoder and Controller Behavioral model Verilog® source code...111
Figure 59. Command Decoder and Controller Structural model Verilog® source code....112

Figure 60. Command Sequence Controller Test Bench Verilog® source code. 113
Figure 61. Command Sequence Controller Behavioral model Verilog® source code. 114
Figure 62. Command Sequence Controller Structural model Verilog® source code. 116
Figure 63. Address Comparator Test Bench Verilog® source code. 117
Figure 64. Address Comparator Behavioral model Verilog® source code. 117
Figure 65. Address Comparator Structural model Verilog® source code. 118
Figure 66. Address Comparator Alternate Structural model Verilog® source code. 119
Figure 67. Address Reference Test Bench Verilog® source code. 120
Figure 68. Address Reference Behavioral model Verilog® source code. 120
Figure 69. Pulse Width Register Test Bench Verilog® source code. 121
Figure 70. Pulse Width Register Behavioral model Verilog® source code. 122
Figure 71. Pulse Width Register Structural model Verilog® source code. 123
Figure 72. Repetition Period Register Test Bench Verilog® source code. 124
Figure 73. Repetition Period Register Behavioral model Verilog® source code 125
Figure 74. Repetition Period Register Structural model Verilog® source code. 126
Figure 75. Tactor Power Controller Test Bench Verilog® source code 127
Figure 76. Tactor Power Controller Behavioral model Verilog® source code. 127
Figure 77. Tactor Power Controller Structural model Verilog® source code. 128
Figure 78. Power Control Logic Test Bench Verilog® source code. 129
Figure 79. Power Control Logic Behavioral model Verilog® source code. 130
Figure 80. Power Control Logic Structural model Verilog® source code. 130
Figure 81. Power Oscillator Test Bench Verilog® source code. 131
Figure 82. Power Oscillator Behavioral model Verilog® source code. 132
Figure 83. Power Oscillator Structural model Verilog® source code. 132
Figure 84. Pulse Width Down Counter Test Bench Verilog® source code. 133
Figure 85. Pulse Width Down Counter Behavioral model Verilog® source code. 134
Figure 86. Pulse Width Down Counter Structural model Verilog® source code. 135
Figure 87. Repetition Period Down Counter Test Bench Verilog® source code 136
Figure 88. Repetition Period Down Counter Behavioral model Verilog® source code 137

Figure 89. Repetition Period Down Counter Structural model Verilog® source code.....137
Figure 90. Clock Divider Test Bench Verilog® source code.....139
Figure 91. Clock Divider Behavioral model Verilog® source code.....140
Figure 92. Clock Divider Structural model Verilog® source code.....140
Figure 93. Clock with Parametric Half-Period Test Bench Verilog® source code. 142
Figure 94. Clock with Parametric Half-Period Behavioral model Verilog® source code...142
Figure 95. D flip-flop Test Bench Verilog® source code. 143
Figure 96. D flip-flop Behavioral model Verilog® source code.....144
Figure 97. Transmission Gate MUX Test Bench Verilog® source code. 145
Figure 98. Transmission Gate MUX Behavioral model Verilog® source code. 145
Figure 99. Structural Schematic for the Twelve-Bit Shift Register. 147
Figure 100. Structural Schematic for the Eight-Bit Data Latch. 148
Figure 101. Structural Schematic for the Input Stream Validity Check. 148
Figure 102. Structural Schematic for the Command Sequence Controller.....149
Figure 103. Structural Schematic for the Address Comparator. 150
Figure 104. Structural Schematic for the Pulse Width Register.....151
Figure 105. Structural Schematic for the Repetition Period Register. 152
Figure 106. Structural Schematic for the Power Control Logic.....153
Figure 107. Structural Schematic for the Power Oscillator.....153
Figure 108. Structural Schematic for the Pulse Width Down Counter.....154
Figure 109. Structural Schematic for the Repetition Period Down Counter. 155
Figure 110. Structural Schematic for the Clock Divider. 156
Figure 111. CMOS PFET and NFET SPICE model definitions.....158
Figure 112. Subcircuits for Fundamental Logic Element SPICE model definitions. 159
Figure 113. Serial Data Receiver SPICE model source code.....164
Figure 114. Serial Data Receiver SPICE model response.....166
Figure 115. Twelve-Bit Input Shift Register SPICE model source code.....167
Figure 116. Twelve-Bit Input Shift Register SPICE model response.....168
Figure 117. Eight-Bit Data Latch SPICE model source code. 169
Figure 118. Eight-Bit Data Latch SPICE model input. 170

Figure 119. Eight-Bit Data Latch SPICE model response. 171
Figure 120. Input Stream Validity Check SPICE model source code. 173
Figure 121. Input Stream Validity Check SPICE model input. 174
Figure 122. Input Stream Validity Check SPICE model response. 175
Figure 123. Command Decoder and Controller SPICE model source code 178
Figure 124. Command Decoder and Controller SPICE model Control response. 180
Figure 125. Command Decoder and Controller SPICE model Register response. 181
Figure 126. Command Sequence Controller SPICE model source code 183
Figure 127. Command Sequence Controller SPICE model response 184
Figure 128. Address Comparator SPICE model source code 186
Figure 129. Address Comparator SPICE model response 187
Figure 130. Pulse Width Register SPICE model source code 189
Figure 131. Pulse Width Register SPICE model response. 190
Figure 132. Repetition Period Register SPICE model source code. 192
Figure 133. Repetition Period Register SPICE model response. 193
Figure 134. Tactor Power Controller SPICE model source code 195
Figure 135. Tactor Power Controller SPICE model response 197
Figure 136. Power Oscillator SPICE model source code. 198
Figure 137. Power Oscillator SPICE model response. 199
Figure 138. Pulse Width Down Counter SPICE model source code. 201
Figure 139. Pulse Width Down Counter SPICE model response. 202
Figure 140. Repetition Period Down Counter SPICE model source code. 203
Figure 141. Repetition Period Down Counter SPICE model response 205
Figure 142. Clock Divider SPICE model source code. 207
Figure 143. Clock Divider SPICE model response 208
Figure 144. Tactile Interface Animation Elements. 210
Figure 145. Tactile Interface Animation in Progress. 211
Figure 146. Intelligent Tactor object JAVA source code. 212
Figure 147. TIC Command object JAVA source code. 216
Figure 148. Tactile Array Demonstration object JAVA source code. 218
Figure 149. Tactile Demonstration Applet HTML source code. 221

Figure 150. Inverter schematic.....223
Figure 151. Two Input NAND Gate schematic. 223
Figure 152. Three Input NAND Gate schematic. 224
Figure 153. Four Input NAND Gate schematic. 224
Figure 154. Three Input AND Gate schematic. 224
Figure 155. Four Input AND Gate schematic. 225
Figure 156. Two Input NOR Gate schematic.....225
Figure 157. Three Input NOR Gate schematic.....225
Figure 158. Two Input XOR Gate schematic.....226
Figure 159. Two Input XNOR Gate schematic.....226
Figure 160. D Flip Flop with Clear schematic. 227
Figure 161. Two Input Multiplexer schematic.....227
Figure 162. Inverter SPICE model source code.....228
Figure 163. Two Input NAND gate SPICE model source code.....228
Figure 164. Three Input NAND gate SPICE model source code.....229
Figure 165. Four Input NAND gate SPICE model source code. 230
Figure 166. Three Input AND gate SPICE model source code.....231
Figure 167. Four Input AND gate SPICE model source code. 232
Figure 168. Two Input NOR gate SPICE model source code.....233
Figure 169. Three Input NOR gate SPICE model source code.....234
Figure 170. Two Input XOR gate SPICE model source code.....235
Figure 171. Two Input XNOR gate SPICE model source code.....236
Figure 172. D Flip Flop with Clear logic element SPICE model source code. 237
Figure 173. Two Input MUX logic element SPICE model source code. 238
Figure 174. Inverter layout.....240
Figure 175. Two Input NAND Gate layout.....241
Figure 176. Three Input NAND Gate layout.....242
Figure 177. Four Input NAND Gate layout. 243
Figure 178. Three Input AND Gate layout.....244
Figure 179. Four Input AND Gate layout. 245
Figure 180. Two Input NOR Gate layout.....246

Figure 181. Three Input NOR Gate layout. 247
Figure 182. Two Input XOR Gate layout. 248
Figure 183. Two Input XNOR Gate layout. 249
Figure 184. D Flip Flop with Clear layout. 250
Figure 185. Two Input Multiplexer layout. 251
Figure 186. Test bench for Parallel Port Data Modulator Verilog® model source code. 254
Figure 187. Test bench for Four-Bit Register Verilog® model source code. 255
Figure 188. Four-Bit Register Verilog® model source code. 256
Figure 189. Test bench for Control State Machine Verilog® model source code. 257
Figure 190. Control State Machine Verilog® model source code. 258
Figure 191. Test bench for Eight-to-One Multiplexer Verilog® model source code. 259
Figure 192. Eight-to-One Multiplexer Verilog® model source code. 260
Figure 193. Four-Bit Register ABEL™ source code. 262
Figure 194. Control State Machine ABEL™ source code. 263
Figure 195. Eight-to-One Multiplexer ABEL™ source code. 266
Figure 196. Four-Bit Register Summary Information. 268
Figure 197. Control State Machine Summary Information. 271
Figure 198. Eight-to-One Multiplexer Summary Information. 274
Figure 199. Command Transmission Driver C++ source code 278
Figure 200. GOMAC Conference Paper (page 1 of 4). 280
Figure 201. GOMAC Conference Paper (page 2 of 4). 281
Figure 202. GOMAC Conference Paper (page 3 of 4). 282
Figure 203. GOMAC Conference Paper (page 4 of 4). 283

LIST OF TABLES

Table 1. Command Word Definitions. 10
Table 2. Address Format Description. 11
Table 3. Pulse Width Register Command Description. 12
Table 4. Repetition Period Register Command Description. 12
Table 5. Required Bus Speed for Different Architectures. 13
Table 6. Intelligent Tactor Input Requirements. 16
Table 7. Summary of Signals for the Twelve-Bit Input Shift Register. 28
Table 8. Summary of Signals for the Eight-Bit Data Latch. 29
Table 9. Summary of Signals for the Input Stream Validity Check. 30
Table 10. Summary of Signals for the Command Sequence Controller. 32
Table 11. Summary of Signals for the Address Comparator. 34
Table 12. Summary of Signals for the Address Reference 35
Table 13. Summary of Signals for the Pulse Width Register. 35
Table 14. Summary of Signals for the Repetition Period Register. 36
Table 15. Summary of Signals for the Power Control Logic. 38
Table 16. Summary of Signals for the Power Oscillator. 39
Table 17. Summary of Signals for the Pulse Width Down Counter. 40
Table 18. Summary of Signals for the Repetition Period Down Counter. 41
Table 19. Summary of Signals for the Clock Divider. 42
Table 20. Comparison of Alternate Logic Designs. 44
Table 21. Inverter Delay Summary 58
Table 22. Component Design Summary. 59
Table 23. Standard Parallel Port Signal Definitions and Pin Assignments. 67
Table 24. Revised Command Structure. 85
Table 25. Legend for Layers used in VLSI Layout. 239

ACKNOWLEDGMENTS

I must first acknowledge the impact God has made on my work over the last 27 months. It is only through his influence that I have been able to remain focused on my research and studies. This work would have been much more difficult without a solid faith upon which to anchor our lives.

My wife Debra and daughter Nickole deserve special recognition and thanks for their unwavering support and understanding. Their love is the glue that binds our family together and has allowed completion of this thesis.

Professor Douglas Fouts was essential in completing this research. He worked hard to secure project funding and continued to support the research when the funding expired. His insight was critical to making the right decisions for this project. I also appreciate his support in getting our work presented at the 1999 GOMAC conference.

Professor Jon Butler was a great instructor and advisor during my study of computer engineering. I share his love of logic problems and his creative insight often sparked some of my most fundamental design decisions.

Dave and Addie Floodeen performed the Herculean task of reviewing this entire thesis for content and flow. Their careful attention corrected many errors and ensured that the words on the paper matched my intended meaning.

Deb Peyton was essential in reviewing the GOMAC paper. Her help with editing supported an outstanding final copy.

Debbie Monroe provided support, encouragement, and project evaluation early in this research. I am grateful for her help and friendship.

Warren Rogers was extremely helpful in manufacturing the circuit board for the Parallel Port Data Modulator.

John Falby provided guidance regarding development of the tactile interface animation. In shaping my approach to software development and coding, he influenced my computer programming skills beyond my expectations.

CDR Angus Ruppert and Brad McGrath of Naval Aerospace Medical Research Laboratory provided excellent input regarding tactile communication physiology. They were essential in defining the tactor timing requirements in the earliest phases of system development.

Finally, I appreciate Terry Wood for his diligent work in developing a miniature computer system that will enable implementation of this tactile interface. His dedication to this project will be the fuel that leads it to success.

I. INTRODUCTION

In a natural environment, people continuously receive sensory information regarding their position and motion from several sources: visual, aural, and somatosensory (distinct bodily sensations such as balance). In an accelerating environment, human sensory information can produce a false sensation of motion. Popular "virtual reality" amusement rides exploit this effect by providing visual, aural, and somatosensory stimulation that generates illusory feelings of motion and acceleration.

Under normal flight conditions, pilots frequently transition between a steady environment and an accelerating one. When most time is spent in a constant velocity condition, all sensory information concurs with the actual motion and spatial orientation is maintained. During periods of extended acceleration, such as steep climbs, the body's somatosensory equilibrium shifts and a false sense of acceleration is experienced during the subsequent steady environment. This conflicting sensory information can cause a loss of situational awareness and spatial disorientation. During periods of visual distraction or obscurity, the pilot must rely on his "feel" for the attitude of the aircraft. The false acceleration sensation has contributed to many aircraft accidents by causing loss of situational awareness and spatial disorientation.

When combing an underwater mine field, divers must swim a geographically referenced search pattern. Geographic position indicators aid the swimmer in combating the effects of current to maintain the desired search pattern. The current guidance system provides a visual display of the required swimming direction. Consequently, the divers'

visual search effectiveness becomes severely degraded while they are referencing their positional displays.

An interface that provides critical information without operator distraction would benefit many military applications. An ideal system would communicate information through a medium that does not interrupt concurrent visual and auditory interchange. A prototypical interface has been developed to tactilely convey information by pulsing tactile transmitters ("tactors"). These tactors are situated around the torso to provide physical stimulus in the form of variable-length, pulsed vibrations.

A. COMMUNICATING THROUGH TOUCH

Touch is a physical sensory input not commonly associated with conveying computer information. Yet, when a person is touched, the response is immediate and often involuntary. The immediate nature of touch response makes it ideal for communicating critical information. Tactile communication can also be the most appropriate interface for specific types of information when existing visual and auditory activities cannot be compromised.

Existing research shows that various sensory responses can be effected by using different tactile stimulus methods. Employing "sensory saltation" can produce a feeling of directional motion using stationary tactors. Using a moving stimulus produces easily interpreted information that is consistent among many observers.[1]

Additional research identified flight information required to properly operate various fighter platforms. The required flight data was evaluated for each of the fighters to determine how well the aircraft presented the parameters to the pilot. Many flight characteristics are poorly represented in each airframe. The research proposed conveying

flight parameters through tactile transmitters mounted in a partial sleeve worn on the pilot's forearm.[10]

B. TACTILE SITUATION AWARENESS SYSTEM BY NAMRL

The Naval Aerospace Medical Research Laboratory (NAMRL) built a rudimentary implementation of tactile communication in their Tactile Situation Awareness System (TSAS). As illustrated in Figure 1, the current TSAS implementation uses a remote, parallel driver to individually power forty (40) tactors. This method requires routing forty pairs of power lines throughout the tactile vest. A simpler communication method is needed to ease vest fabrication and maintenance. Additionally, the microprocessor is constantly burdened with directly controlling power application to every individual tactor.

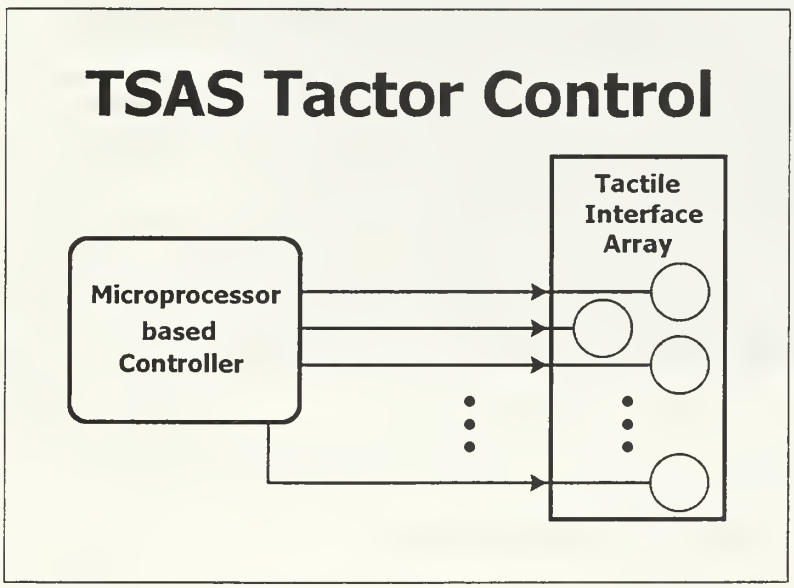


Figure 1. TSAS Tactor Control.

The wiring harness requirements could be dramatically reduced by using a bus communication structure with local power switching. This approach would provide a standardized wiring scheme and eliminate the continuous processor load caused by remote

power control. A bus architecture would also maximize flexibility by allowing the number of tactors to be varied between interface applications.

A miniaturized network interface card will allow connecting all tactors to a single information bus as shown in Figure 2. Each interface chip will continuously monitor the bus for a command addressed to its tactor. Upon detection of a properly addressed command packet, the interface card will decode and execute the command. Power will be switched by the interface chip to allow the controlling microprocessor to dedicate its full processing ability to interfacing with the host technology and determining the best tactile representation of the received platform parameters.

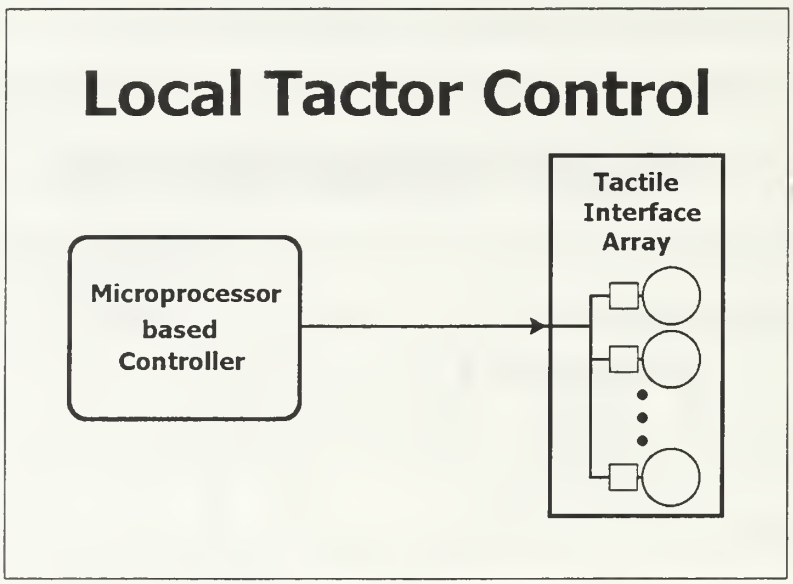


Figure 2. Local Tactor Control.

C. DEVELOPING INTELLIGENT TACTORS

To refine the tactile interface, we developed a compact communication topology for connecting each tactile transmitter to the controlling microprocessor. Serial communications were selected for this application to minimize the number of conductors required for data transfer.

An application-specific Tactor Interface Chip (TIC) provides the necessary hardware to realize the communication scheme. Each tactor in a forty-element array will include a TIC, as shown in Figure 3, that controls tactor activation. This hardware combination forms an "intelligent tactor" that shifts waveform creation from the microprocessor to the individual tactor assemblies. The resulting decrease in computational load allows use of a slower microprocessor, decreasing system power consumption.

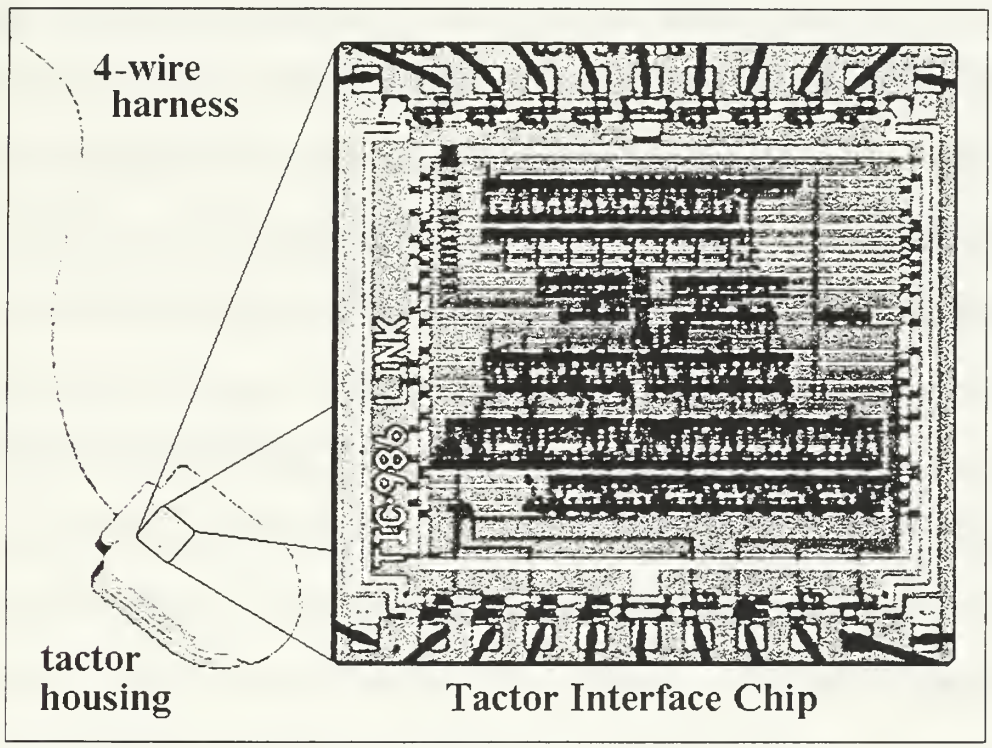


Figure 3. Composition of the Intelligent Tactor.

D. THESIS OUTLINE

The remainder of this thesis is organized as follows. Chapter II discusses development of the communication protocol. Chapter III discusses specification and design of the Tactor Interface Chip. Chapter IV discusses the layout and evaluation for the VLSI implementation of the chip. Chapter V describes development of a parallel-port data modulator used to drive the tactor array during testing and demonstration. Chapter VI

discusses testing the fabricated chip. Chapter VII describes revisions to the communication protocol. Chapter VIII discusses design changes to incorporate additional features in the interface chip. Finally, Chapter IX contains conclusions and suggestions for future work.

Many appendices are included to provide specific technical data necessary to fully understand the design efforts and decisions. Appendix A contains listings of the Verilog source code used to design and evaluate the electronic modules that comprise the TIC. Appendix B provides schematic diagrams of all TIC modules and components. Appendix C details the SPICE simulations performed to validate and verify all aspects of the TIC design. Appendix D covers the animation program used to illustrate the operational relationships between the received components and the various TIC components. Appendix E contains the design details of the VLSI logic elements used to implement the TIC on a single microchip. Appendix F provides all design efforts in creating the Parallel Port Data Modulator for sending commands from a standard computer parallel port to the tactile array. Appendix G documents the program written in C++ to place command bytes on the parallel port for subsequent transmission by the command modulator. Appendix H includes a copy of Reference 2, the paper presenting this research to the 1999 Government Microcircuit Application Conference (GOMAC). The files listings from all appendices have been compiled separately on CD-ROM.

II. COMMUNICATION PROTOCOL

To support implementation of a tactile information interface, it is necessary to develop a communication protocol that meets the system control requirements. When a suitable command protocol is defined, various architectures can be evaluated to determine the best option for rapid communication between a controlling microprocessor and (at least) forty tactile transmitters. Flexibility and expansion are supported by using a common communication bus and intelligent tactors. As mentioned in the previous chapter, an intelligent tactor is formed by mounting a Tactor Interface Chip (TIC) in the tactor housing (see Figure 3) to locally control tactor activation. This chapter presents the command structure developed and the communications architectural design to implement the tactile interface.

A. DESIGN REQUIREMENTS

To establish a framework for system design, we must first take a macroscopic view of the intelligent tactor. Fundamentally, the TIC must control the application of current to the attached tactor as directed by the system controller. Additionally, the command structure must support the addition of ore tactors to the present system.

1. Required Output

Each TIC must provide a controlled, bi-directional current to the attached tactor in response to commands it receives from the controlling microprocessor. When activated, the TIC must energize the tactor at the specific frequency for which the tactor is designed. The activation duty cycle is determined by the commands received; commands are fully discussed in the following section. The TIC must activate the tactor as soon as an

appropriate command is received and it must immediately stop tactile stimulus when a "terminate" command is received.

Bi-directional current is achieved using the switching network illustrated in Figure 4. The TIC activates the switch pairs of Figure 4 in an alternating fashion to drive current through the tactor in opposite directions.

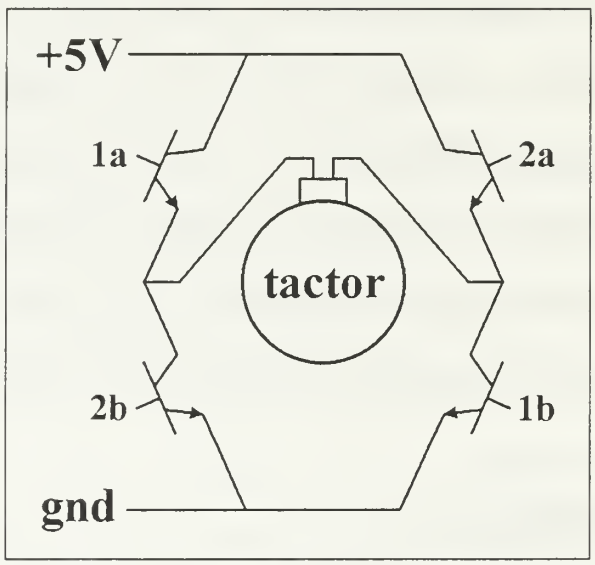


Figure 4. Tactor Current-Switching Structure.

2. System Configuration

The tactor control system must be capable of independently issuing commands to forty individual tactors. The activation cycle should repeat with a minimum period of 100 mS and a maximum of 4000 mS. During the activation cycle, the TIC must be able to adjust the length of activation from a minimum of approximately 50 mS to a maximum of approximately 1000 mS. Finally, the system must be able to sequentially activate two tactors within one millisecond (1 mS) of each other.

B. CONTROL STRUCTURE

1. Address

Transmission of tactile messages requires each tactor in the forty-element array to be capable of producing defined pulse shapes. These tactile signals can be independent or synchronized with several other tactors. Each TIC must be able to recognize commands meant to control its attached tactor since the pulse shape parameters are transmitted on a common data bus. Unique identification is accomplished by assigning an "address" to each TIC.

2. Pulse Shape

Tactors are repeatedly pulsed to convey information to the user. Changing the pulse duration and pulse rate creates different physical sensations; this can be used to relate differing messages. Pulse shape production requires two parameters, pulse width and repetition period, illustrated in Figure 5. These values are stored in TIC data registers and are used to control tactor activation.

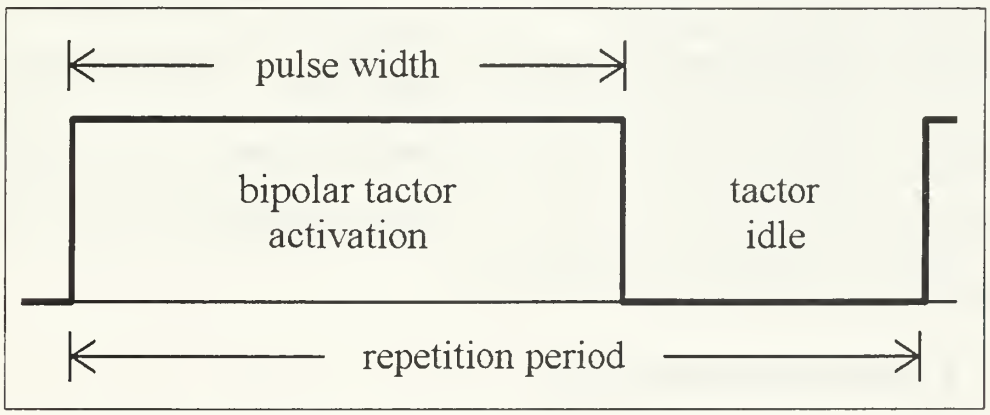


Figure 5. Tactor Activation Parameters.

C. COMMAND FORMAT

It is best to use an eight-bit command language format, if possible, since the tactor array is being driven by a commercial-off-the-shelf microprocessor and associated communication ICs. Therefore, the first iteration of the command structure evaluated the feasibility of incorporating all desired TIC functionality within the 256 different eight-bit commands. To simplify the interpretation of the commands, it is most effective to group the commands together in a way that minimizes the number of bits that uniquely identify a command type. Considering these two goals and the three basic command types it is best to separate the commands into one set of 128 and two sets of 64. To accommodate the desired number of tactors and to allow for future expansion with more tactors or multiple addresses on a single TIC (discussed later), the 128 command group is assigned to the address commands. This provided similar command words for the two register-type command sets, pulse width and repetition period. The command distribution plan is outlined in Table 1 and discussed in more detail in the following subsections.

Command Word	Meaning
0 x x x x x x x	7-bit Address
1 0 x x x x x x	6-bit Pulse Width
1 1 x x x x x x	6-bit Repetition Period

Table 1. Command Word Definitions.

1. Address Command Word

The tactor address command word indicates to which tactor the subsequent command is being sent. This addressing plan allows issuing a command to a single tactor since each TIC contains a unique identifier. Table 2 summarizes the address command

word format. The all-zeros address is not used since this is the reset condition of the TIC input register and internal data bus. The current convention provides capacity for up to 126 tactors. Additionally, the format can support future incorporation of tactor group addressing. The all-ones address is hard-wired into every TIC to provide a universal command capability. Uses for the "ALL CALL" configuration include turning off all tactors during operations or energizing all tactors for testing. System design allows stringing several addresses together before issuing the register command bytes. This will facilitate concurrently issuing an identical command to numerous tactors.

Address Word	Meaning
0 0 0 0 0 0 0 0	Reserved -- TIC bus idle condition
0 0 0 0 0 0 0 1 to 0 1 1 1 1 1 1 0	Addresses for up to 126 tactors; may also include group addresses.
0 1 1 1 1 1 1 1	ALL CALL -- all tactors respond

Table 2. Address Format Description.

2. Pulse Width Word

The Pulse Width command sets the actual length of time the tactor is energized. The Pulse Width command format is summarized in Table 3. This command is implemented by passing a value that represents the number of 16 mS time divisions to apply power to the tactor. A value of zero is used to turn off the tactor. Using 16 mS time divisions with a 6-bit multiplier (factor) produces 63 possible activation lengths including 0, 16, 32, . . . and 1008 mS. The 16 mS time divisions are generated by dividing an input reference pulse.

Register Command	Meaning
1 0 0 0 0 0 0 0	Turn tactor OFF
1 0 0 0 0 0 0 1 to 1 0 1 1 1 1 1 1	Set Pulse Width to 1 through 63 multiples of 16 mS (16 to 1008 mS)

Table 3. Pulse Width Register Command Description.

3. Repetition Period Word

The Repetition Period command defines the period used for pulse repetition. The Repetition Period command format is summarized in Table 4. This command is implemented by passing a value that represents the number of 64 mS time divisions to wait before re-energizing the tactor. If Pulse Width is greater than zero, a zero Repetition Period will energize the tactor continuously. Using 64 mS time divisions with a 6-bit multiple produces 63 possible repetition period lengths ranging from 64 mS to 4032 mS. The 64 mS time divisions are also generated by dividing an input reference pulse. A repetition value that represents a time length less than or equal to the "on" time will keep the tactor continuously energized.

Register Command	Meaning
1 1 0 0 0 0 0 0	Tactor ON continuously if PW > 0
1 1 0 0 0 0 0 1 to 1 1 1 1 1 1 1 1	Set Repetition Period to 1 through 63 multiples of 64 mS (64 to 4032 mS)

Table 4. Repetition Period Register Command Description.

D. BUS ARCHITECTURAL CONSIDERATIONS

A primary concern regarding interface design is ease of system fabrication and maintenance. Basic error detection is necessary from an operational perspective to prevent

system response to spurious noise. A parity checksum is used to detect single-bit errors. This section compares two communications architectural design options for the tactile interface.

1. Command Promulgation Speed

From a system architecture perspective, the most critical constraint is the speed at which commands must be implemented by the TIC. This constraint is extrapolated from the specification for a 1 mS maximum time between commands. The 1 mS maximum command separation requirement can be met using either a parallel or serial data bus by adjusting the data transmission clock speed. To evaluate the minimum data bus speed for different architectures, the frequency required to transmit a given command length in 1 mS is calculated. In a parallel implementation, each command byte requires a single clock cycle to transmit. In a serial implementation, each byte requires eleven clock cycles: a start bit, eight data bits, a parity bit, and a stop bit. Table 5 summarizes the bus speed requirements.

Command Length	Parallel Bus Speed	Serial Bus Speed*
2 bytes	2 kHz	22 kHz
3 bytes	3 kHz	33 kHz
4 bytes	4 kHz	44 kHz
5 bytes	5 kHz	55 kHz
* Serial communication incurs a 3-bit overhead for data packet formatting.		

Table 5. Required Bus Speed for Different Architectures.

2. Parallel Bus

Parallel bus architecture allows the fastest data transfer from the microprocessor to the TIC. However, Table 5 shows that data transfer rates are not a limiting factor for this

application since bus speeds over 1 MHz are available. The advantages and disadvantages of using parallel bus architecture include:

a. Advantages:

(1) Simplified TIC circuit design. The TIC could directly latch the data byte from the external bus onto the internal command bus.

(2) Much faster data transfer or lower required bus speed for a given data rate. The reduction in required speed would reduce the required transmission power.

b. Disadvantages:

(1) Additional wiring is required in the harness assembly and vest for data communications. This greatly complicates the fabrication process and makes maintenance and repair much more difficult. This also increases the size of the wiring harness and the weight of the system implementation.

3. Serial Bus

Serial bus architecture reduces the number of wires needed for data transferring but it requires a much more complex TIC input design. Advantages and disadvantages of using serial bus architecture include:

a. Advantages:

(1) A minimum number of wires can be used in the wiring harness and vest. This will ease fabrication and maintenance while reducing the size and weight.

b. Disadvantages:

(1) Much slower data transfer rate or higher required bus speed for a given data rate.

(2) Much more complex TIC input circuitry. A serial to parallel decoder must be implemented to support conversion of the serial data stream to parallel command words.

E. ARCHITECTURAL DECISION AND JUSTIFICATION

The Serial Bus architecture is used for this implementation. This choice is primarily based on the following essential considerations:

1. Fewer harness conductors will make vest construction and maintenance much easier. Fewer connections at each TIC will also reduce the risk of failure and incorrect wiring. Additionally, fewer wires will minimize the system size and weight.

2. Conversion from parallel data to a serial communication stream is easy to include at the controlling microprocessor. This custom parallel-to-serial conversion can be easily adapted to allow use of many different microprocessors for future implementations.

3. Serial to parallel conversion at the TIC can be included in the VLSI design and actually requires about the same layout area necessary to accommodate eight additional input pads.

F. TRANSMISSION PACKET FORMAT

The Universal Synchronous/Asynchronous Receiver-Transmitter (USART) standard provides a format for transmitting eight bit data by encapsulating the data into an eleven-bit packet. The USART packet model is used to package the command bytes into a serial bit stream that can be easily detected. The command packet, illustrated in Figure 6, includes a start bit, eight data bits, a parity bit, and a stop bit. This package format also provides basic fault protection by detecting all single-bit errors. While idle, the data line is held at a logic "1" (+5 V).

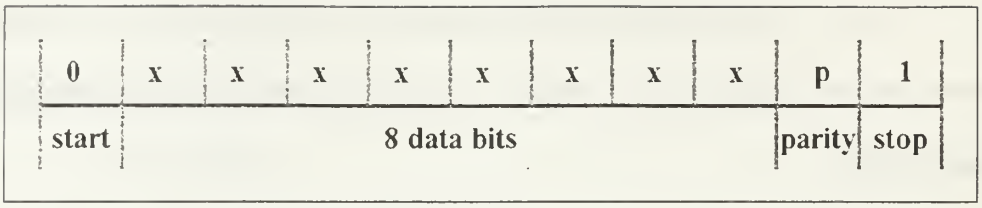


Figure 6. Standard USART data packet format.

When no data is present, the serial communication line is in an idle state, held at logic 1. The data packet begins with a start bit that consists of a single 0. The start bit is followed by eight bits of data, which are transmitted in order from the most significant bit to the least significant bit. A running count is performed on the data bits and the number of 1's is used to calculate the value of the parity bit. Using odd parity (the typical mode) yields a parity bit of 1 when the data bit count is even and a value of 0 when the data bit count is odd. This scheme always produces an odd number of 1's at the receiver when the eight data bits and single parity bit values are counted. The parity bit provides detection of all single-bit errors in the data stream. Finally, a stop bit of 1 is sent and the system is ready to transmit the next data packet.

G. PHYSICAL CONSTRUCTION REQUIREMENTS

Inputs to the TIC fall into two basic categories: chip/tactor power and data/timing signals. Table 6 summarizes the input requirements for the intelligent tactor.

Line		Description
a)	+5 V	Power for TIC and tactor
b)	Ground	Common ground line
c)	Data	Serial communication bus
d)	Clock	Synchronous clock signal

Table 6. Intelligent Tactor Input Requirements.

1. TIC and Tactor Power

The chip and tactor share a single +5 V power supply and a single ground line. Very little power is needed to operate the chip since CMOS FET technology was used for TIC fabrication. The entire chip, consisting of roughly 2000 transistors, requires approximately 8 mA of current. Each tactor will require between 100 mA and 250 mA (depending on installed tactor) during operation.

2. Command Data and Timing Signals

The TIC receives all data on a single line whose voltage is referenced to the common ground. A clock signal provided by the microprocessor facilitates synchronous serial data transfer. The clock signal is also used to generate the timing references for the control down counters and the tactor current oscillator.

III. TACTOR INTERFACE CHIP SPECIFICATION AND DESIGN

After defining the communication protocol needed to convey control signals to the tactile interface, it is necessary to develop the hardware that will convert the serial commands into tactile stimulus. The Tactor Interface Chip (TIC) is the application-specific integrated circuit that converts the serial command stream into the bi-directional current that drives the tactile stimulators. After discussing the TIC design goals, this chapter relates the development process through all levels of abstraction. The conceptual operation is first discussed as a system that is broken into three functional modules. Each functional module is then defined by its operational requirements. The functional modules are then separated into several assemblies with specific, cardinal tasks. Behavioral level system modeling and simulation is then explained. Behavioral model conversion into logical structures and the simulation and testing is described next. Finally, advanced system design features included in the TIC are discussed.

A. DESIGN GOALS

In creating an intelligent tactor, the two primary design goals resulted from the need to incorporate the TIC directly into the tactor casing. First, to reduce the size of the tactor casing, all control circuitry must fit onto a single VLSI control chip. Second, to simplify tactile interface production, a single design must be used for all tactors in the array. In the prototypical version, use of a single TIC design for all tactors is possible by externally setting the address parameter by grounding TIC input pins. In a future implementation, a better mechanism could be devised to define the address of an individual tactor. The initial TIC design also does not incorporate the solid-state power switches necessary for causing bi-directional current to flow in the tactor. External power transistors are used to provide

current switching based on control signals from the TIC. This design was accepted due to the expense of fabricating large transistors in a BiCMOS chip. Planned modifications to the existing design are discussed in the "Future Improvements" section of this thesis.

B. OPERATIONAL CONCEPT

Conceptually, the Tactor Interface Chip will interpret commands received on a serial data bus and control tactor activation based on those commands. This scheme can be broken into the three functional areas, recovery of the eight-bit command from the serial data stream, interpretation of the command to affect wave shape parameters, and generation of the ordered waveform. This organization is illustrated in Figure 7 and discussed in the following subsections.

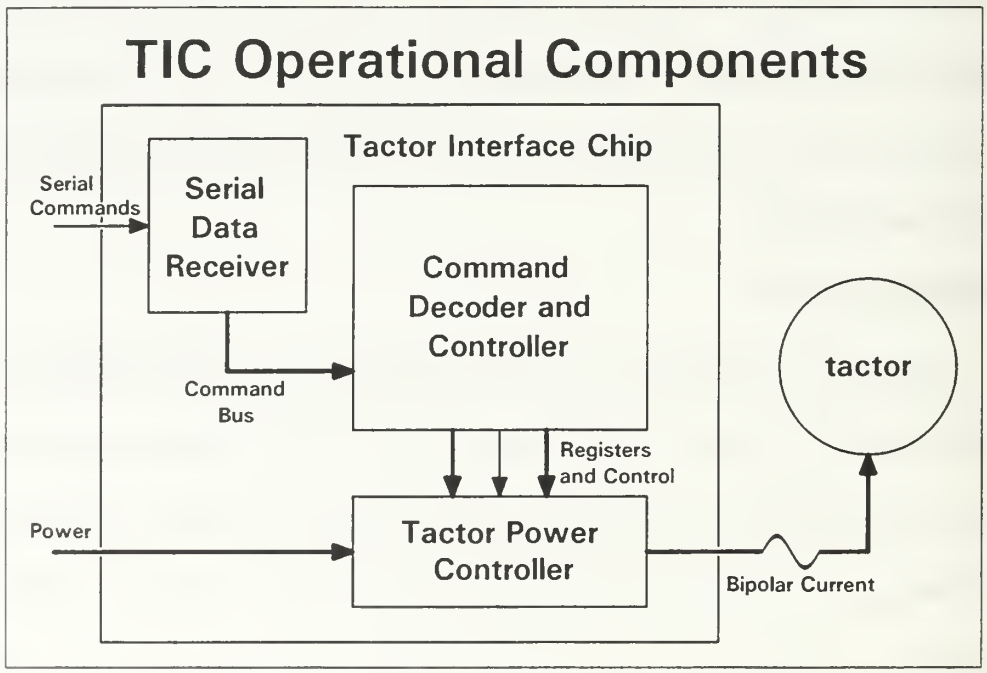


Figure 7. TIC Functional Modules.

1. Serial Data Receiver

The first functional component, the Serial Data Receiver, continuously monitors the serial data bus and decodes the bit stream to detect properly formatted data packets. When a valid packet is detected, the command byte is latched onto an internal command bus and a data-valid signal is sent to the Command Decoder and Controller.

2. Command Decoder and Controller

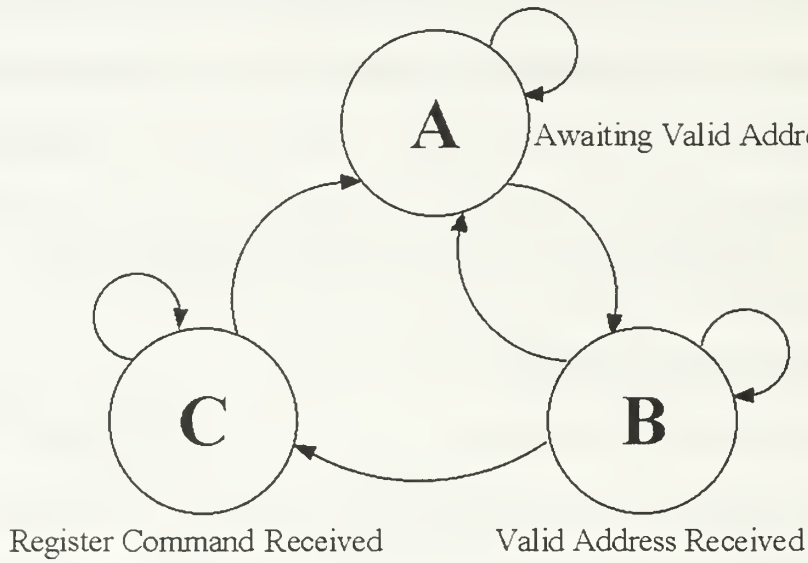
The second functional component is the Command Decoder and Controller. This module interprets every command received to determine if the command applies to the attached tactor. Relevant commands are executed and the associated memory registers are updated. Extraneous commands are ignored. The Tactor Power Controller is notified of changes to ensure that tactor activation is immediately adjusted to conform to the new parametric settings.

3. Tactor Power Controller

The final functional component is the Tactor Power Controller. This module continuously produces two complementary timing signals tuned to the operating frequency of the attached tactor. These signals are used to alternately activate the switch pairs in the current-switching network shown in Figure 4. When timing signals are applied to the switching network, the tactor provides stimulus to the user. The activation wave shape described in Figure 5 is created by passing and blocking the oscillation signals based on the Pulse Width and Repetition Period values stored in the memory registers of the Command Decoder and Controller.

C. OPERATIONAL DESCRIPTION

The heart of TIC operations is the Command Decoder and Controller. As new commands are placed on the command bus by the Serial Data Receiver, the data-valid signal triggers evaluation by the Command Decoder and Controller. Response to the received commands is controlled by the existing TIC operational state. The operational state changes based on the current state and the valid commands received by the TIC. The state diagram in Figure 8 illustrates the TIC operating sequence and defines the state transitions.



Initial State	Transition Condition	Next State
All	Reset asserted	**A**
A	Bus valid & valid address	**B**
B	Any other address received	**B**
B	Any register command received	**C**
C	Any address received	**A**
C	Any register command received	**C**

Figure 8. TIC Operating States and Transitions.

Initially, the TIC is in a monitor state waiting to receive a valid address. When an appropriate address is received, the TIC shifts to a condition that waits for a command to set the register values. When a register command is received, the TIC enters a state that responds to all register commands until an address is detected. Any address received after a register command marks the end of the command cycle and shifts the TIC to the monitor state where it waits for the next properly addressed command set. This operating sequence provides easy control consistent with the defined communication structure. An additional benefit of this approach is that it allows a set of register commands to be sent to several tactors simultaneously by preceding the commands with a string of address words.

D. FUNCTIONAL MODULE DESIGN

The top-down approach greatly simplifies circuit design by separating the specific tasks into three functional modules. Each functional module is designed to operate independently with well-defined inputs and outputs. This modular approach also greatly simplifies testing at all design levels.

1. Serial Data Receiver

The Serial Data Receiver (Figure 9) continuously monitors the input data line to detect and latch transmitted packets onto the command bus. It consists of a twelve-bit shift register, a validity checker, and an eight-bit latch. The twelve most recent data bits received on the serial data input line are stored in the shift register. The entire 12-bit set is evaluated using the data-packet format rules. When a string of bits is detected that meets the validity check, the byte embedded within the data packet is latched onto the command bus. The latch signal also generates a "Bus Data Valid" signal that triggers command decoding. When a valid command is latched, a feedback path partially clears the shift register. This

clearing action ensures that two immediately sequential data packets do not produce an erroneous command detection as shown in Figure 10. The partial clearing action also resets the latch signal since the shift register contents no longer match the required data packet format.

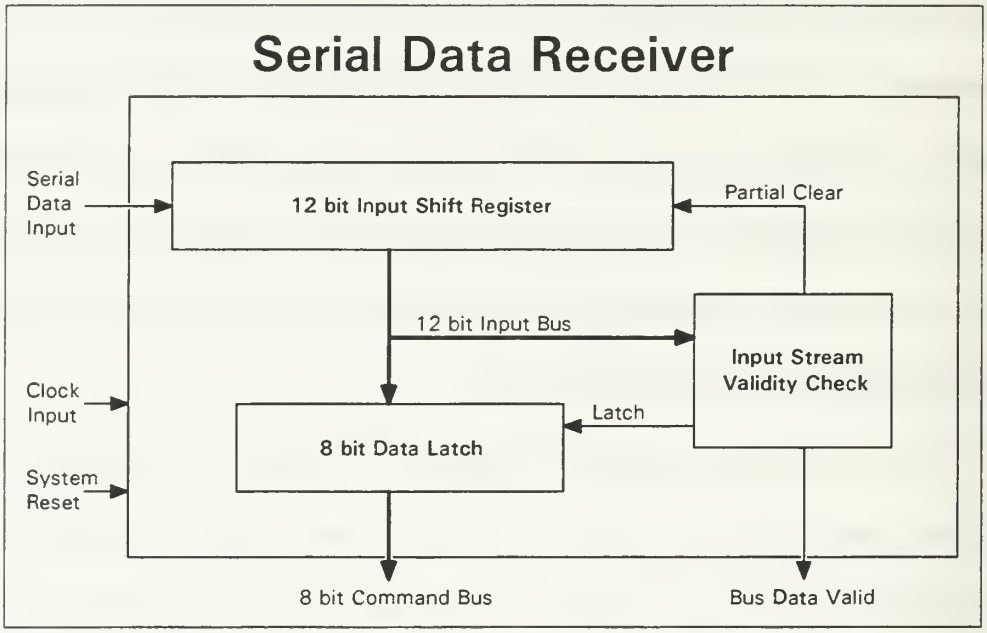


Figure 9. Serial Data Receiver Elements.

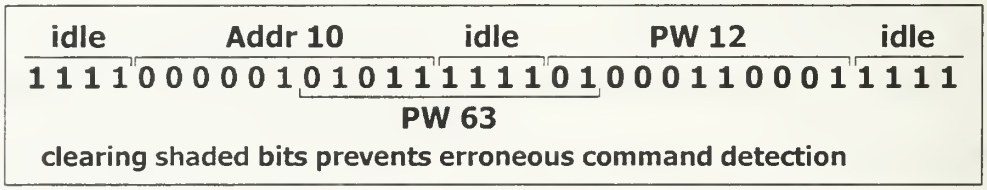


Figure 10. Partial clearing prevents Erroneous Command Detection.

2. Command Decoder and Controller

The Command Decoder and Controller (Figure 11) evaluates the received commands and adjusts the internally stored waveform parameters if the command is properly addressed to the attached tactor. It consists of a sequence controller, address comparator, address reference, and two six-bit registers. The sequence controller is a state

machine (refer to Figure 8) that causes the TIC to react only to properly addressed commands. The address reference maintains a unique address for the individual tactor. The address comparator provides a "valid address" signal if the command bus holds either the value stored in the address reference or the "all call" address. The TIC ignores all received commands until the address comparator detects a valid address. It then updates the stored value of pulse width or repetition period with every new register command. The pulse width and repetition period registers operate identically. The registers continuously monitor the command bus and indicate when the register value matches the bus value. If a command is received that attempts to set the register to its current value, the command is ignored to prevent a spurious interruption of the tactor activation cycle. If the difference signal indicates that the register value must be changed, the new value is latched and the difference signal is used to clear the latch signal. When the register value is updated, a control signal is generated to force the Tactor Power Controller to restart the tactor activation cycle so it will match the new register values. This resetting action ensures that received register commands are instantly implemented, thus securing tactor activation immediately upon receipt of a termination (set pulse width to zero) command. Subsequently, when an address is received, the TIC returns to the monitor condition and waits for the next properly addressed command set.

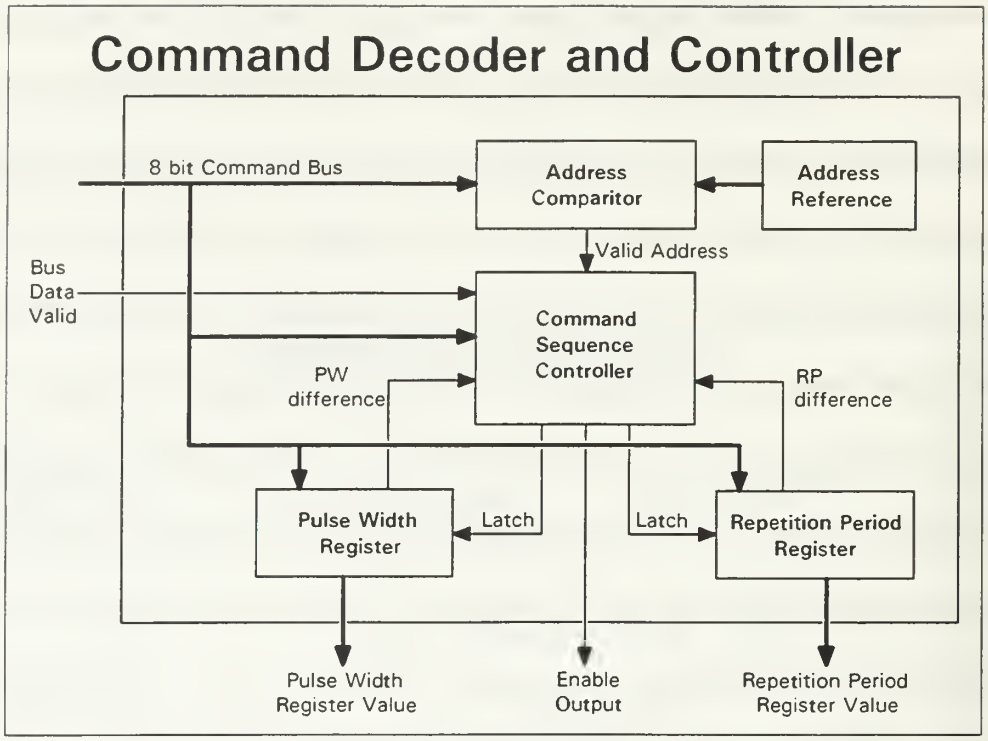


Figure 11. Command Decoder and Controller Elements.

3. Tactor Power Controller

The Tactor Power Controller (Figure 12) converts the input data signals into pulsed bi-directional current that is applied directly to the tactor by using the switching network illustrated in Figure 4. A frequency divider reduces the 1 MHz clock to a selectable tactor oscillating frequency and a 62.5 Hz down counter clock. The oscillator frequency is applied to the power oscillator to produce alternating current for the tactor. The power controller uses two synchronized down counters to create the stored wave. Both down counters are designed to count once from the loaded value to zero, maintaining the zero value once it is reached. The pulse width down counter includes a status signal indicating when the count value is equal to zero. The repetition period down counter includes a status signal indicating when the count is greater than one. The control logic clears or loads both down counters based on the down counter conditions and the control signal received from the Command

Decoder and Controller. These two down counter conditions are used to control tactor activation by either passing or blocking the oscillation signals to the current switching network.

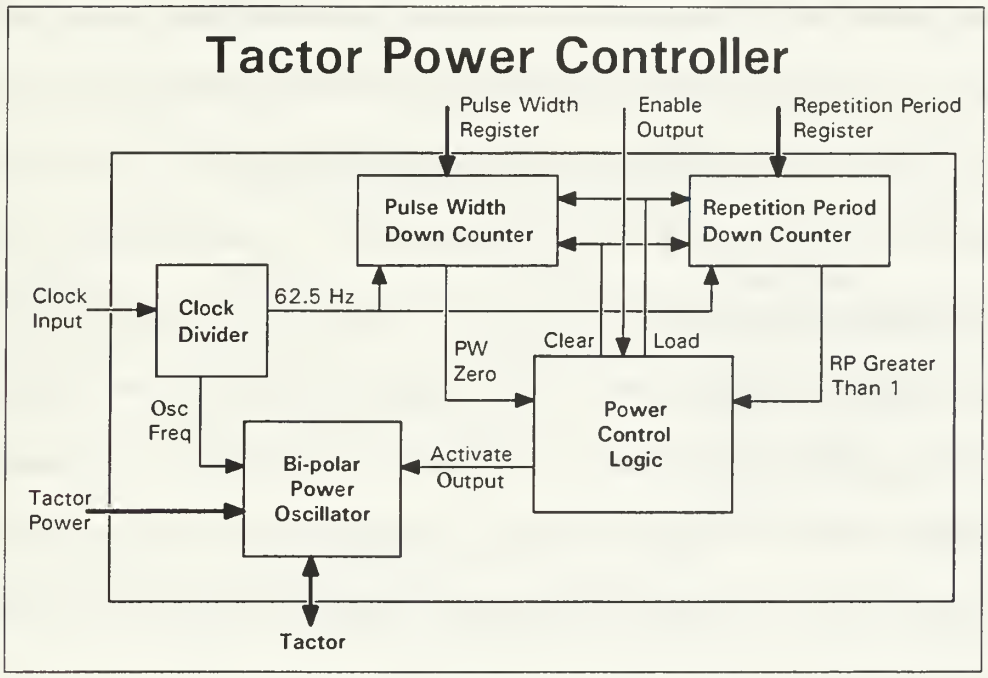


Figure 12. Tactor Power Controller Elements.

E. VERILOG® DESIGN VERIFICATION

Each functional module was simulated and thoroughly tested using the Verilog® modeling system. First, behavior models were designed for all components and tested to validate the design descriptions. The components were then assembled to create the functional modules and tested to ensure proper operation of each module. The functional modules were then assembled into a behavioral model for the entire TIC. This system model was fully tested to ensure proper operation of the entire interface before implementing the behavioral definitions into structural elements. The Verilog® models and testing "benches" used for system design are included in Appendix A.

1. Twelve-Bit Input Shift Register

The twelve-bit input shift register is critical to decoding the serial data stream into the transmitted command bytes. Twelve bits are required to validate the input stream because the data packet format is eleven characters long and because the data line is held at a logic "1." The shift register accepts input from the serial data line, clock, reset, and partial clear. The shift register provides an output bus containing the value for each of the most recent twelve bits received by the Serial Data Receiver. Table 7 summarizes the signals used and produced by the twelve-bit input shift register. On the rising edge of each clock cycle, a new data bit is latched into the lowest position of the shift register and all other bits are shifted up one position. When a reset signal is received, all bits on the output bus are immediately cleared to a logic "0." When a partial clear signal is received, the oldest ten bits on the output bus are immediately cleared to a logic "0" and the lowest two bits retain their existing values.

Twelve-Bit Input Shift Register Input and Output			
Input		Output	
Signal	Source	Signal	Destination
input data	serial data input	input bus	data latch, input validity check
partial clear	input validity check		
clock	clock input		
reset	system		

Table 7. Summary of Signals for the Twelve-Bit Input Shift Register.

2. Eight-Bit Data Latch

The eight-bit data latch drives the internal TIC command bus. The data latch receives input from the shift register output, reset, and a latch signal. Input from the shift

register is limited to the data lines representing the data-packet command byte. The output from the data latch is simply the TIC command bus value. Table 8 summarizes the signals used and produced by the eight-bit data latch. When a latch signal is received, the data latch locks the value of each command bit onto the command bus. When a reset signal is received, all bits on the command bus are immediately cleared to a logic "0."

Eight-Bit Data Latch Input and Output			
Input		Output	
Signal	Source	Signal	Destination
input bus*	shift register	command bus	command sequence controller, address comparator, pulse width register, repetition period register
latch	input validity check		
reset	system		
* only the bit positions representing the command byte			

Table 8. Summary of Signals for the Eight-Bit Data Latch.

3. Input Stream Validity Check

The input stream validity check component continuously evaluates the shift register output to detect a properly formatted command packet. The validity checker receives input from the shift register output, reset, and clock. The validity checker produces the latch signal used by the data latch, the partial clear signal used by the shift register, and a bus valid signal used by the command sequence controller component of the Command Decoder and Controller module. Table 9 summarizes the signals used and produced by the input stream validity checker.

Input Stream Validity Check Input and Output			
Input		Output	
Signal	Source	Signal	Destination
input bus	shift register	latch	data latch
clock	clock input	bus valid	command sequence controller
reset	system	partial clear	shift register

Table 9. Summary of Signals for the Input Stream Validity Check.

A format flag is used to create the output latch signal. The format indicator is continuously generated by evaluating the twelve bits input from the shift register against the packet format rules. Four specific conditions are required to generate the format flag: a) the highest bit must be a logic "1" representing either an idle serial input bus or the stop bit from a previous command, b) the second highest bit must be a logic "0" representing the start bit for the current command, c) the lowest bit must be a logic "1" representing the stop bit for the current command, and d) the parity check must yield a logic "1." The parity check is performed using an XOR of the data bits and the parity bit. The format command may experience some perturbations immediately following the positive clock edge as the input bits change because the shift register data is shifted on the rising clock edge. To avoid creating an erroneous latch command, the latch signal is not generated until the second half of the clock cycle.

When the latch signal is triggered, a bus valid signal is produced to indicate the presence of a valid command to the Command Decoder and Controller. This bus valid flag continues for ten clock cycles, at which time it is cleared. The signal that clears the bus valid flag is produced by the bus valid signal and the detection of "0 1" in the highest two bit positions of the shift register. This condition can only exist when the stop bit from the

current valid command reaches the second highest position of the shift register. Elimination of the bus valid signal prepares the command bus for the command byte that might immediately follow the current command.

After the format flag and the clock produce the latch signal, the command byte is locked onto the TIC command bus. Then, a partial clear signal clears the highest ten bits in the shift register to prevent the shifting bits from producing an erroneous command detection. The partial clear is generated by detecting both the format flag and the bus valid signal at the same time; a condition that indicates a valid command has been successfully latched. When the partial clear signal clears the highest ten input bits, the format becomes incorrect and the format flag becomes "0." When the format flag changes "0," the latch condition is lost and the latch signal is reset to "0." Additionally, the partial clearing of the shift register forces the "0 1" transition that will clear the bus valid signal in ten clock cycles.

4. Command Sequence Controller

The command sequence controller is the component in the Command Decoder and Controller module that acts as the central command processor for the TIC. It is responsible for interpreting received commands and establishing the ordered tactile stimulus parameters. The command sequence controller receives the bus valid signal, the valid address signal, the pulse width difference flag, repetition period difference flag, clock, and reset signal. The command sequence controller generates an enable output signal used by the Tactor Power Controller module, a pulse width latch signal for the pulse width register, and a repetition period latch signal for the repetition period register. Table 10 summarizes the signals used and produced by the command sequence controller.

Command Sequence Controller Input And Output			
Input		Output	
Signal	Source	Signal	Destination
command bus	data latch	enable output	power control logic
pulse width difference	pulse width register	pulse width latch	pulse width register
repetition period difference	repetition period register	repetition period latch	repetition period register
bus valid	input validity check		
valid address	address comparator		
clock	clock input		
reset	System		

Table 10. Summary of Signals for the Command Sequence Controller.

When the system reset is applied, the command sequence controller shifts to state 0 and the enable output, pulse width latch, and repetition period latch signals are all set low. Since all outputs from this component are dependent on the controller operational state, the state transitions are discussed before the actual output signals are described. With the exception of a system reset, all output parameters are changed only after both a valid address has been detected and a register command has been received. Additionally, the outputs may change many times during the period following a valid register command.

The command sequence controller remains in state 0 until a valid address is present on the command bus. The valid address signal from the address comparator allows the command sequence controller to shift to state 1. The command sequence controller remains in state 1 for all subsequent address commands until a register command (either pulse width or repetition period) is present on the command bus. The register command causes the

sequence generator to shift to state 3. State 2 is a transient condition that is not used by the command sequence controller and any occurrence of state 2 resets to state 0. The command sequence controller remains in state 3 until an address is present on the command bus. The address command and bus valid signal causes the command sequence controller to shift to state 0. Notably, if the address command is also a valid address, the command sequence controller will shift from state 0 to state 1 on the next clock cycle.

As mentioned above, all output signal adjustments (except system reset) are made only when the command sequence controller is in state 3. This discussion refers to a command that sets either register generically as a "register command" since the pulse width and repetition period registers operate identically. When a valid register command sets the pulse width to zero, the command sequence controller immediately clears the enable output signal, thus immediately stopping tactile stimulus. Typically, a register command is received that does not match the current register value. This register mismatch triggers two actions to occur simultaneously. First, it clears the enable output signal, causing the Tactor Power Controller to clear its counters in preparation for a change in wave shape parameters. In addition, the register latch signal is produced, ordering the appropriate register to lock the commanded wave parameter into the storage register. As soon as the new register value has been latched, the register-difference flag changes to indicate the values now match. This change in the register-difference flag clears the latch signal and sets the output enable signal, causing the Tactor Power Controller to restart wave shape generation with the new wave shape parameters. In the event that a register command is received that matches the current register value, the latch signal is not needed and the output enable signal does not cycle.

Effectively, the matching register command is ignored since it provides no change to the current operating condition.

5. Address Comparator

The address comparator provides indication when the valid command byte is an address command that matches either the address reference or the all call address. Input to the address comparator comes from the command bus and the address held in the address reference. The address comparator produces the valid address signal that is used by the command sequence controller. Table 11 summarizes the signals used and produced by the address comparator.

Address Comparator Input And Output			
Input		Output	
Signal	Source	Signal	Destination
command bus	data latch	valid address	command sequence controller
address	address reference		

Table 11. Summary of Signals for the Address Comparator.

6. Address Reference

The address reference maintains the address assigned to the attached tactor. The input source for the prototypical implementation comes from external jumper connections on the TIC. The output is a buffered reflection of the settings. Table 12 summarizes the signals used and produced by the address reference.

Address Reference Input And Output			
Input		Output	
Signal	Source	Signal	Destination
input address	address input pads	address	address comparator

Table 12. Summary of Signals for the Address Reference.

7. Pulse Width Register

The pulse width register stores the most recent pulse width setting received by the TIC. This setting is used by the Tactor Power Controller module to create the stimulus waveform. The pulse width register receives input from the command bus, a latch signal from the command sequence controller, and the system reset. The register produces an output pulse width value used by the pulse width down counter and a pulse width difference signal used by the command sequence controller. Table 13 summarizes the signals used and produced by the pulse width register.

Pulse Width Register Input And Output			
Input		Output	
Signal	Source	Signal	Destination
command bus*	data latch	pulse width value	pulse width down counter
latch	command sequence controller	pulse width difference	command sequence controller
reset	system		
* only the bit positions representing the embedded register value			

Table 13. Summary of Signals for the Pulse Width Register.

The pulse width register continuously provides a difference signal indicating when the stored six-bit value is different from the lowest six bits on the command bus. This output is used by the command sequence controller to regulate the latch signal and to reset

waveform generation by the Tactor Power Controller. When a latch signal is received from the command sequence controller, the pulse width register locks the timing value held in the lowest six bits of the command bus into its storage register. When latching is complete, the difference signal indicates a match, providing a feedback signal to the command sequence controller. When a reset signal is received, the pulse width register clears all output bits to a logic "0."

8. Repetition Period Register

The repetition period register stores the most recent repetition period setting received by the TIC. This setting is used by the Tactor Power Controller module to create the stimulus waveform. The repetition period register receives input from the command bus, a latch signal from the command sequence controller, and the system reset. The register produces an output repetition period value used by the repetition period down counter and a repetition period difference signal used by the command sequence controller. Table 14 summarizes the signals used and produced by the repetition period register.

Repetition Period Register Input And Output			
Input		Output	
Signal	Source	Signal	Destination
command bus*	data latch	repetition period value	repetition period down counter
latch	command sequence controller	repetition period difference	command sequence controller
reset	system		
* only the bit positions representing the embedded register value			

Table 14. Summary of Signals for the Repetition Period Register.

The repetition period register continuously provides a difference signal indicating when the stored six-bit value is different from the lowest six bits on the command bus. This output is used by the command sequence controller to regulate the latch signal and to reset waveform generation by the Tactor Power Controller. When a latch signal is received from the command sequence controller, the repetition period register locks the timing value held in the lowest six bits of the command bus into its storage register. When latching is complete, the difference signal indicates a match, providing a feedback signal to the command sequence controller. When a reset signal is received, the repetition period register clears all output bits to a logic "0."

9. Power Control Logic

The power control logic component controls implementation of the tactile stimulus. Input to the power control includes an enable output signal, a pulse-width-equals-zero signal, and a repetition-period-greater-than-one signal. The power control logic produces an enable power signal, a clear counter signal, and a load counter signal. Table 15 summarizes the signals used and produced by the power control logic.

Power Control Logic Input And Output			
Input		Output	
Signal	Source	Signal	Destination
enable output	command sequence controller	enable power	power oscillator
pulse width equals zero	pulse width down counter	clear counter	pulse width down counter, repetition period down counter
repetition period greater than one	repetition period down counter	load counter	pulse width down counter, repetition period down counter

Table 15. Summary of Signals for the Power Control Logic.

When the enable input signal from the command sequence controller is not enabled, the power control logic provides a clear signal to both down counters. The clear signal locks both counters at zero. When a pulse-width-equals-zero signal is present, the power control logic maintains the enable-power at logic "0," causing the power oscillator to block the oscillation signals from reaching the current switching network. When the repetition-period-greater-than-one signal is low, the power control logic signals the down counters to reload the stored register values on the next clock cycle.

10. Power Oscillator

The power oscillator controls transmission of the signals that drive the switches in the current switching network. Input to the power oscillator is an enable output signal and the oscillation frequency. The power oscillator produces the signals that enable the current switching network to create the bi-directional current for the attached tactor. Table 16 summarizes the signals used and produced by the power oscillator.

Power Oscillator Input And Output			
Input		Output	
Signal	Source	Signal	Destination
enable power	power control logic	power switch set 1	current switching network
oscillation frequency	clock divider	power switch set 2	current switching network

Table 16. Summary of Signals for the Power Oscillator.

The enable power signal controls the passage or blocking of the oscillation signals. When enable power is high, the power oscillator passes the oscillation frequency signals to the current switching network causing the tactors to vibrate. When enable power is low, the oscillation signals are blocked and there is no tactile stimulation. The oscillation frequency is complemented to provide alternating signals that produce the bi-directional switching characteristic needed for the current switching network.

11. Pulse Width Down Counter

The pulse width down counter provides the timing that defines the activation interval for the tactor. Input to the pulse width down counter is the stored pulse width value, a load counter signal, a clear counter signal, and the counter clock. The pulse width down counter produces the pulse-width-equals-zero signal. Table 17 summarizes the signals used and produced by the pulse width down counter.

Pulse Width Down Counter Input And Output			
Input		Output	
Signal	Source	Signal	Destination
pulse width value	pulse width register	pulse width equals zero	power control logic
count clear	power control logic		
count load	power control logic		
clock	clock divider		

Table 17. Summary of Signals for the Pulse Width Down Counter.

When the count clear signal is present, the pulse width down counter immediately clears its count to zero and produces the pulse-width-equals-zero signal. The count clear signal takes precedence over all other inputs. When the count load signal is present, the counter loads the value that is stored in the pulse width register on the next clock cycle. The clock signal is provided by the 62.5 Hz output of the clock divider.

12. Repetition Period Down Counter

The repetition period down counter provides the timing that defines the repetition interval for the tactor. Input to the repetition period down counter is the stored repetition period value, a load counter signal, a clear counter signal, and the counter clock. The repetition period down counter produces the repetition-period-greater-than-one signal. Table 18 summarizes the signals used and produced by the repetition period down counter.

Repetition Period Down Counter Input And Output			
Input		Output	
Signal	Source	Signal	Destination
repetition period value	repetition period register	repetition period greater than one	power control logic
count clear	power control logic		
count load	power control logic		
clock	clock divider		

Table 18. Summary of Signals for the Repetition Period Down Counter.

When the count clear signal is present, the repetition period down counter immediately clears its count to zero and clears the repetition-period-greater-than-one signal. The count clear signal takes precedence over all other inputs. When the count load signal is present, the counter loads the value that is stored in the repetition period register on the next clock cycle. The clock signal is provided by the 62.5 Hz output of the clock divider.

13. Clock Divider

The clock divider uses a 14-stage counter to generate the frequency signals for tactor oscillation and proper down counter timing. Input to the clock divider is from the system clock and the reset signal. Output from the clock divider includes signals at 250 Hz, 125 Hz, and 62.5 Hz. The 1 MHz system clock is divided by two at each stage of the clock divider. Table 19 summarizes the signals used and produced by the clock divider.

Clock Divider Input And Output			
Input		Output	
Signal	Source	Signal	Destination
clock	clock input	62.5 Hz	pulse width down counter, repetition period down counter
reset	System	125 Hz	power oscillator
		250 Hz	power oscillator

Table 19. Summary of Signals for the Clock Divider.

F. STRUCTURAL COMPONENT DESIGN

After completing the behavioral design, each component was converted to a circuit using logic gates. Most of the design conversion was very straightforward using the digital design techniques described in Reference 1. To ensure proper operation of the state machine in the command sequence controller, many of the advanced state-machine design techniques presented in Reference 8 were used.

The structural design process was performed in two distinct steps, basic structural implementation and component optimization for minimum power and size. The initial structural designs were fully tested using Verilog® models; the model source code is included in Appendix A. The optimized structural designs were iteratively developed and tested using SPICE. The final circuit schematics are included in Appendix B and the SPICE models are included in Appendix C.

The circuit optimization techniques are illustrated below and the design of each component is described in the following subsections. Specific optimization efforts are discussed with each affected component.

1. Structural Circuit Optimization

Two methods used to reduce the power consumption and physical layout area are logic function minimization and negative logic. Function minimization uses logic analysis to determine the smallest sum-of-products equation to realize a given function. Negative logic results in reduced power consumption by reducing the total number of transistors. Both techniques require less layout area on the microcircuit since they result using fewer, smaller logic components. Figure 13 illustrates two possible structural implementations of the address comparator. The circuit on the left uses conventional logic while the circuit on the right makes extensive use of negative logic. Table 20 summarizes the reduction in layout area, power consumption, and propagation delay. A very noticeable advantage of using negative logic is the improved signal propagation speed. The improved response time comes primarily from reducing the number of transistors in the signal path.

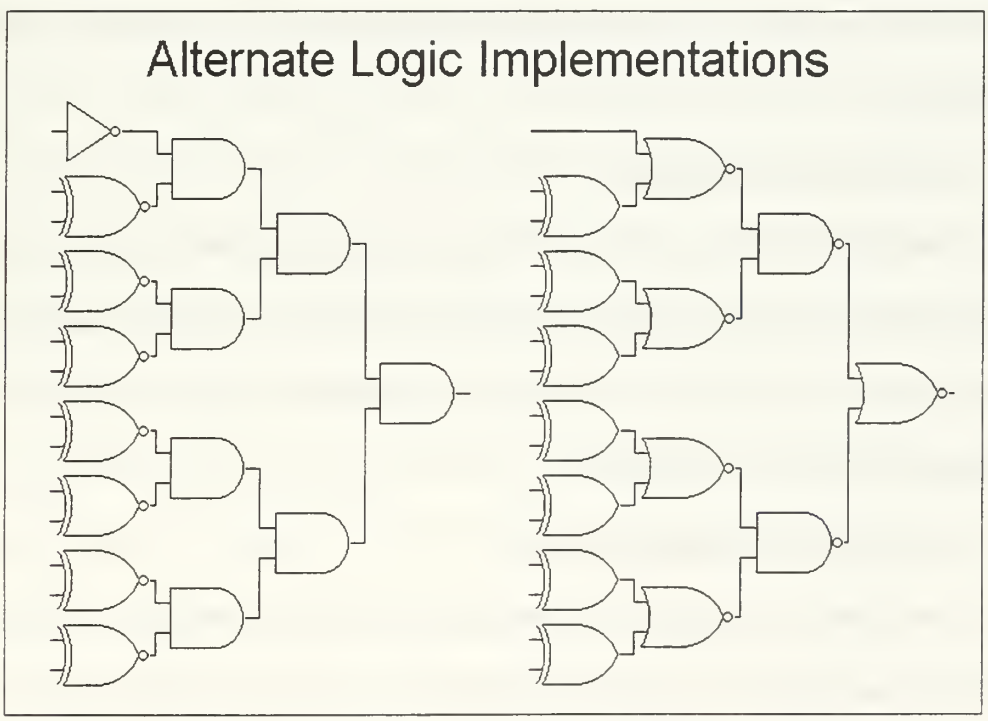


Figure 13. Alternate Structures for Realizing the Address Comparator.

	Area [μm^2]	Transistors	Delay [nS]*
Conventional Logic	29,500	128	2.96
Negative Logic	25,900	112	1.88
Difference	3,600	16	1.08
% Difference	12.2%	12.5%	36.5%
* Delay time was calculated using SPICE to simulate the actual CMOS FET implementation of each circuit.			

Table 20. Comparison of Alternate Logic Designs.

2. Twelve-Bit Input Shift Register

The twelve-bit shift register consists of twelve D-flip/flops wired in series. All flip/flops share a common clock signal that is driven directly from the input clock signal. The system reset drives the clear signal for the lowest two D-flip/flops. The highest ten flip/flops are cleared by either the system reset or the partial clear signal that comes from the input validity checker. The schematic for the optimized circuit is included as Figure 99 of Appendix B.

3. Eight-Bit Data Latch

The eight-bit data latch consists of eight D-flip/flops wired in parallel. All flip/flops share a common latch signal provided by the input validity checker. The system reset drives the clear signal for all of the D-flip/flops. The schematic for the optimized circuit is included as Figure 100 of Appendix B.

4. Input Stream Validity Check

The input stream validity checker consists of a packet-format section, latch-signal driver, bus-data-valid driver, and partial-clear driver. The packet-format section determines if the stream of input bits is properly formatted by ANDing the two stop bits, the parity

indicator, and the complement of the start bit. The parity indicator is simply an XOR of all data bit positions with the parity bit position. The latch signal is created when the format is correct and the clock signal is low. The bus data valid signal is driven by a D-flip/flop that stores the format signal on the latch signal upward transition. The bus data valid signal is cleared by either the system reset or when a "0 1" logic combination is detected in the two highest bit positions of the shift register and the bus data valid signal is set. The partial clear signal is generated when the packet format is correct and the bus data valid signal is set.

The schematic for the optimized circuit is included as Figure 101 of Appendix B. In hindsight, it may have been more efficient to use an S/R latch for the data bus valid signal. This option was not considered during the design evaluation but should be attempted in the next TIC version.

5. Command Sequence Controller

The command sequence controller consists of a state machine, two register latch drivers, and the enable output S/R latch. The operating state is stored in a pair of D-flip/flops. The logic that drives the state machine was derived using transition and output analysis detailed in Chapter 7 of Reference 8. The implemented transition logic prevents hazards and race conditions. The design also recovers from any occurrence of the undefined state 2. The pulse width and repetition period latch signals are driven by the receipt of a register command that does not match the currently stored value. The enable output latch is set any time the state machine is in state 3 and there is a valid register command on the command bus. The enable output latch is reset for any change in the pulse width or repetition period register values. The enable output signal is also immediately cleared when

a system reset is detected or when a valid, zero pulse width command is detected. The schematic for the optimized circuit is included as Figure 102 of Appendix B.

6. Address Comparator

The structural design for the address comparator consists of an all call detector and an equality test for the reference address. The all call detector simply indicates when the all call command, "0 1 1 1 1 1 1 1," is present on the command bus. The address equality test uses an XNOR for each bit position to determine if the command bus matches the provided address reference. The schematic for the optimized circuit is included as Figure 103 of Appendix B.

7. Address Reference

The structural design for the address reference in this initial prototype consists simply of buffers from the TIC input pins. No schematic diagram is included for this component.

8. Pulse Width Register

The structural design for the pulse width register consists of six D-flip/flops wired in parallel and an equality test to indicate a difference between the command bus and the stored pulse width value. All flip/flops share a common latch signal provided by the command sequence controller. The system reset drives the clear signal for all of the D-flip/flops. The equality test uses an XOR for each bit position to determine if the command bus differs from the stored pulse width value. The schematic for the optimized circuit is included as Figure 104 of Appendix B.

9. Repetition Period Register

The structural design for the repetition period register consists of six D-flip/flops wired in parallel and an equality test to indicate a difference between the command bus and the stored repetition period value. All flip/flops share a common latch signal provided by the command sequence controller. The system reset drives the clear signal for all of the D-flip/flops. The equality test uses an XOR for each bit position to determine if the command bus differs from the stored repetition period value. The schematic for the optimized circuit is included as Figure 105 of Appendix B.

10. Power Control Logic

The structural design for the power control logic consists of a single inverter. This component routes the appropriate signals between other components in the TIC circuit. The output from the two down counters was modified to eliminate the need to invert their signals before sending them to the other components. The schematic for the optimized circuit is included as Figure 106 of Appendix B.

11. Power Oscillator

The structural design for the power oscillator consists of a dual-path switch and four signal amplifiers. The dual-path switch will either block or pass the oscillation frequency and its complement. When the oscillation signal is passed, the four drivers provide sufficient current to activate the switch pairs in the current switching network. The schematic for the optimized circuit is included as Figure 107 of Appendix B.

12. Pulse Width Down Counter

The structural design for the pulse width down counter consists of six D-flip/flops configured as a down counter and a comparison circuit to indicate when the counter value is

not equal to zero. The down counter has an input selector that shifts operation between down counting and loading the value stored in the pulse width register. All flip/flops share a common clock signal provided by the clock divider as a timing reference. The counter output is continuously tested to indicate when the value is not equal to zero. The counter reset circuit includes a function that stops the down counter when it reaches zero. The counter is also reset by the count clear signal from the power control logic. The schematic for the optimized circuit is included as Figure 108 of Appendix B.

13. Repetition Period Down Counter

The structural design for the repetition period down counter consists of eight D-flip/flops configured as a down counter and a comparison circuit to indicate when the counter value is not greater than one. The down counter has an input selector that shifts operation between down counting and loading the value stored in the repetition period register. When loaded, the repetition period value is stored in the upper eight bits of the counter and the lowest two bits are set to zero. All flip/flops share a common clock signal provided by the clock divider as a timing reference. The combination of the two extra bit positions and the same clock frequency causes the repetition period down counter to operate at a time interval that is four times longer than that of the pulse width down counter. The counter output is continuously tested to indicate when the value is not greater than one. The counter reset circuit includes a function that stops the down counter when it reaches zero. The counter is also reset by the count clear signal from the power control logic. The schematic for the optimized circuit is included as Figure 109 of Appendix B.

14. Clock Divider

The structural design for the clock divider consists of fourteen D-flip/flops configured as an up counter. All flip/flops share a common clock signal that is driven directly from the input clock signal. The up counter continuously acts as a frequency divider that provides two oscillation frequencies and the down counter timing reference. The available oscillation frequencies are 250 Hz and 125 Hz. The down counter timing reference is 62.5 Hz. The system reset drives the clear signal for all of the D-flip/flops. The schematic for the optimized circuit is included as Figure 110 of Appendix B.

G. ADVANCED DESIGN FEATURES

Several features of the initial TIC design provide enhanced system performance. The first two features were included as enhancements to the minimum design specification because they provide improved functionality while coupling easily with the conceptual operations. The third advanced feature, an onboard reset, was included to ensure that the TIC establishes a consistent initial condition when it is first energized. The final two features were added for chip testing and evaluation with various tactile transmitters.

1. Multiple Command Packet Addressing

The operating-state transition definitions allow a command byte stream that includes multiple TIC addresses. This feature allows a register command to be transmitted to several tactors simultaneously. This capability can be used to reduce the volume of data transmitted on the serial communication wire from the micro-controller. When commands are sent to numerous tactors in this fashion, all tactors activate with a single, synchronized wave-shape.

2. All-Call Address

The command value "0 1 1 1 1 1 1 1" is reserved as an All-Call address that produces a valid address response for all TICs. This feature is primarily intended to enable rapid termination of all tactile stimuli. This address can also be used to test the entire communication array.

3. Dual Reset Circuit

The analog response of the CMOS circuit components is used to produce an initial reset signal for the first 200 nS of TIC operation. The initial reset forces all TIC elements to establish a consistent condition when the circuit is energized. Included in the system-reset circuit is a selectable, low-voltage reset. This reset element is included to protect the system from erratic response caused by low input voltage. The low voltage feature can be disabled using an external TIC jumper, if necessary, for circuit testing.

4. Selectable Oscillator Frequency

An input jumper allows selection between the two tactor-oscillation frequencies: 125 Hz and 250 Hz. This selection allows the TIC to be used with different tactors. These two frequencies were selected because they are available in the clock divider and because they reasonably match the input requirements of the tactors being considered for use in the prototype system.

5. Selectable Address

By including the TIC address as an external input, a single TIC design can be used for all tactors in the communication array. This feature allows the greatest flexibility for prototype testing, since it allows a single "intelligent tactor" to function in every possible

array position. A similar approach will likely be used in future versions to limit production and inventory requirement to a single TIC/tactor assembly.

H. ANIMATION OF TACTOR INTERFACE CHIP OPERATIONS

As a tool to explain TIC response to command bytes, an operating animation was developed that illustrates the changes that occur in the TIC registers and counters as a string of commands is received. A more detailed explanation of the animation program is contained in Appendix D.

1. TIC Visual Representation

Figure 14 shows the graphical representation of two intelligent tactors in a tactile array. The dark gray rectangles represent the tactors. Each is labeled with its address value. The number at the bottom of each column represents the register value for the pulse width or repetition period. The column represents the value in the down counter associated with each register. The horizontal bar across the bottom represents simulation time and proceeds steadily from left to right. The rectangular bubbles above the time line are commands that will be issued when the time reaches their position.

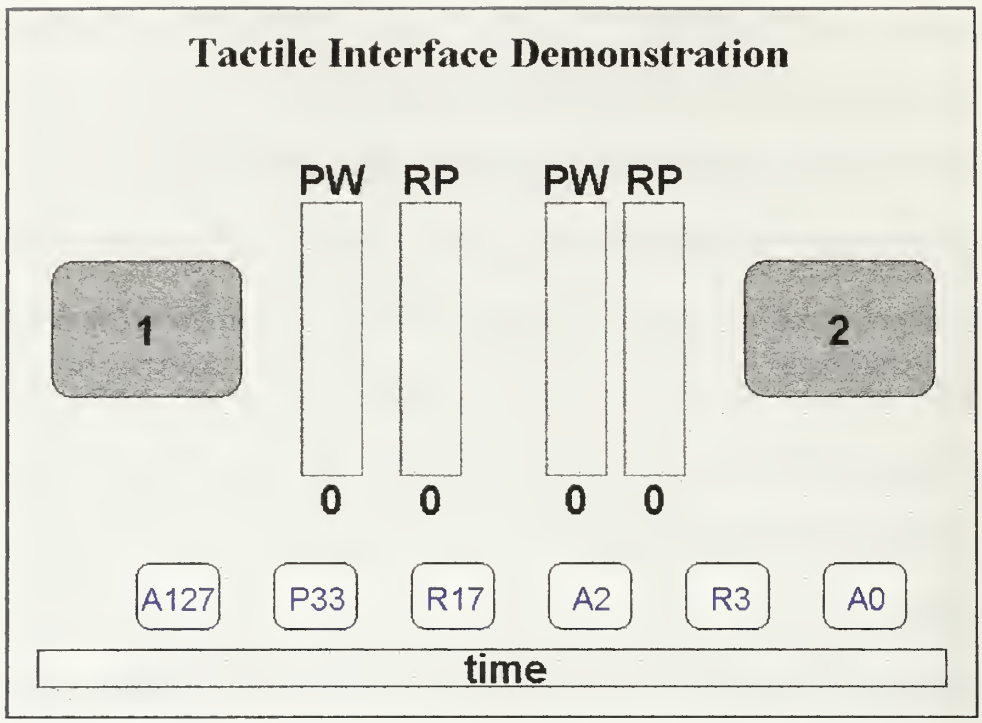


Figure 14. Tactile Interface Animation Basics.

2. Animation Color Scheme

Figure 15 shows the animation in progress. The tactor rectangles change in color to represent the state of the Command Decoder and Controller. When a valid address is received, the TIC shifts to state "B" and the tactor color changes to yellow. When a register command is received by a tactor in state "B," the register is set to the commanded value, the TIC shifts to state "C," and the tactor color changes to red. When any address is received by a tactor in state "C," the TIC shifts to state "A" and the tactor color changes back to gray. The pulse width down counter value is represented by a green column in the area below the "PW" label. The repetition period down counter value is represented by a blue column in the area below the "RP" label. During operation, the green column drops four times as fast as the blue column. As long as the green pulse-width column is not zero, the associated tactor vibrates. When the green pulse-width column reaches zero, the vibration stops.

When the blue column is not greater than one, both down counters load the stored register value. Thus, when a zero repetition period is assigned, the pulse width column reloads on every clock pulse and does not decrease in value. When the repetition period register is greater than zero, both counters decrease until they are reloaded at the repetition period down counter value of one.

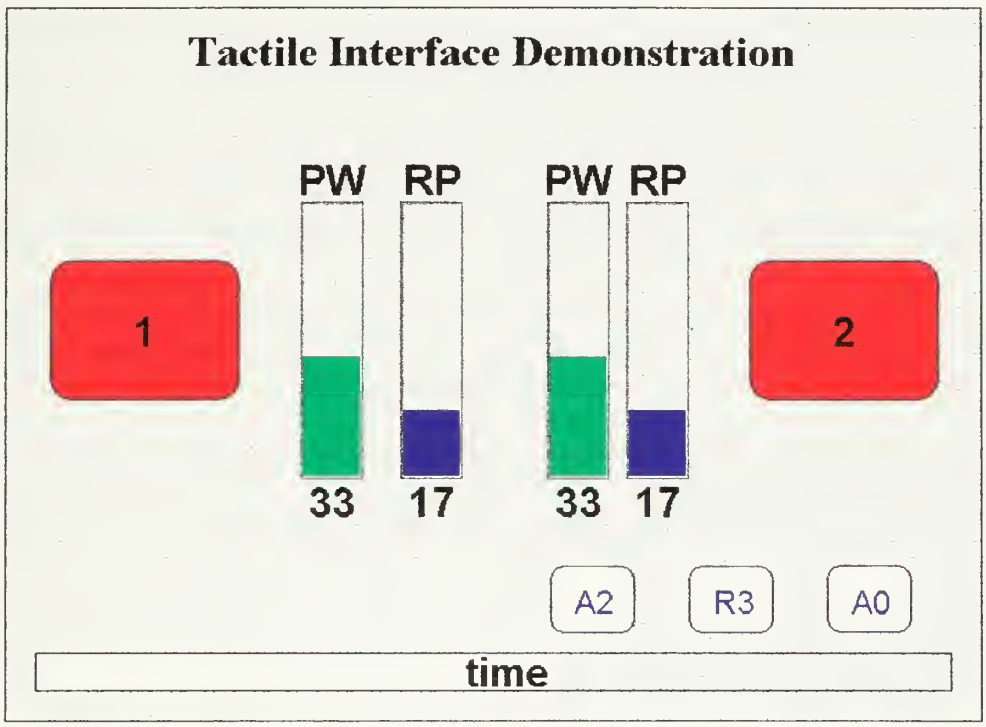


Figure 15. Tactile Interface Animation in Progress.

IV. TACTOR INTERFACE CHIP VLSI IMPLEMENTATION

In preparation for the VLSI implementation of the Tactor Interface Chip, the specific system priorities must be examined. These priorities then work together to define the CMOS FET size. After determining the optimum FET size for the application, the elements must be built to support basic logic functions. These logic elements are then combined to form the larger components. The components are then assembled into the functional modules and, finally, into the composite system. After the chip input and output are added, the chip is ready for comprehensive testing and fabrication. This chapter covers the entire VLSI implementation process.

A. COMPETING VLSI DESIGN CONSTRAINTS

During VLSI design, many requirements are juxtaposed. High speed transistors are physically larger and consume more power. Conversely, minimum sized transistors require the least amount of power but their speed may be insufficient when driving large interconnect lines or numerous down-stream components. These factors were balanced to meet the design requirements of the Tactor Interface Chip.

1. Size

Funding limits forced microchip size to be a primary constraint. Chip size primarily bounds the number of circuit components because component interconnections consume the majority of VLSI layout area. This sharply limited circuit complexity and fundamentally affected many of the design decisions made in the previous chapter.

2. Power

Each TIC must draw minimum current from the battery-powered system since the tactile interface is a stand-alone bridge between the information source and the human user.

Using the smallest possible CMOS FETs throughout the circuit minimizes total power consumption. Aggressively simplifying the logic structure further reduced power requirements while increasing response speed.

3. Speed

Small transistor size adversely influences response time. Minimum transistor size is sufficient at a 1 MHz clock speed unless long interconnects or several components must be driven. Individual elements were resized based on their output loading.

B. CMOS FET TRANSISTOR SIZING

1. Determining PFET Size From NFET Size

Based primarily on power considerations, minimum size CMOS FETs were examined to determine their suitability for the TIC application. Beginning with the absolute minimum transistor width of 3 μm, the response of an inverter was evaluated. When sizing FETs the mobility of the charge carriers must be considered. PFET width must be significantly larger that NFET width to balance system output since the majority carrier for NFETs are electrons (high mobility) and the majority carriers for PFETs are holes (low mobility). Figure 16 shows the effect of using different size PFETs with a minimum size NFET. The ideal sizing produces an inverted output of 2.5 volts as the input sweeps through 2.5 volts. By examining Figure 16, a PFET width of 7 μm most closely achieves the ideal condition. However, a more conventional PFET width to NFET width ratio of 2.0 was used for this VLSI layout; making PFETs 6 μm wide. Checking the response in Figure 16 shows that a width of 6 μm is still very close to the ideal response.

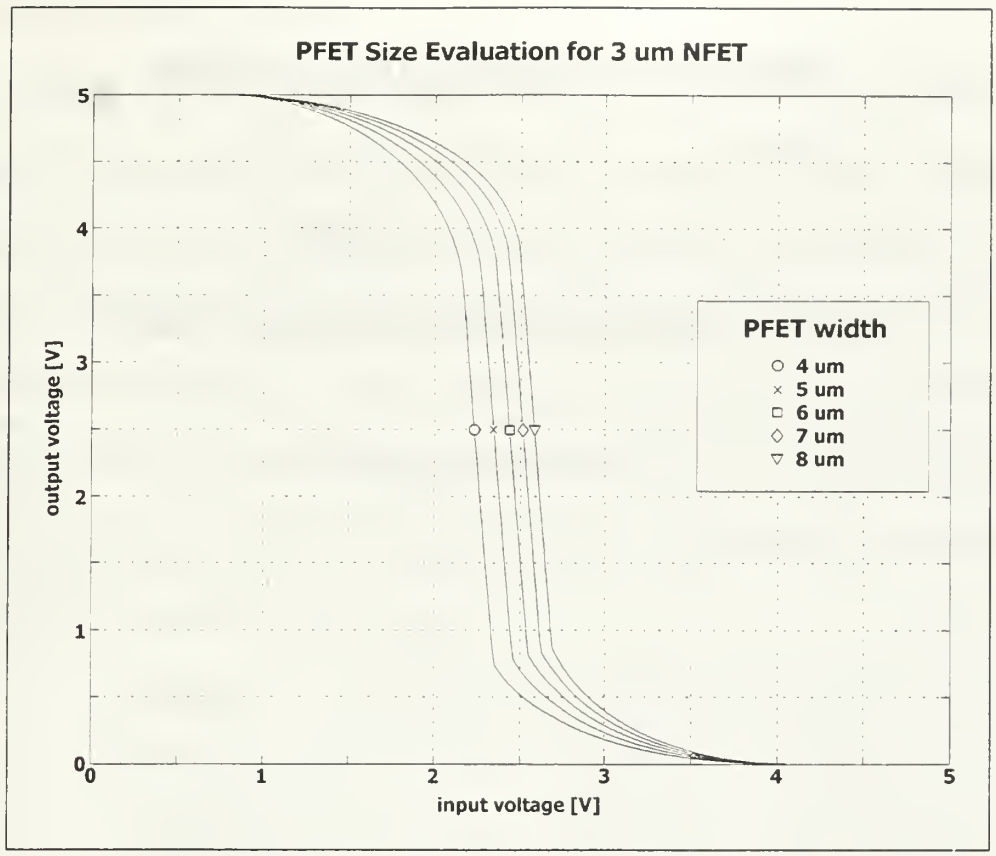


Figure 16. Inverter Response for Various PFET Widths.

2. Basic Response Timing

Having determined the PFET size needed with a minimum size NFET, response timing had to be verified to ensure the system would transmit the signals quickly enough to support a 1 MHz clock speed. This measurement was accomplished by simulating a series of inverters and measuring transmission delay between two inverters near the series end as illustrated in Figure 17. Figure 18 shows the inverter response for the delay circuit and Table 21 provides the actual delay values calculated from the simulation.

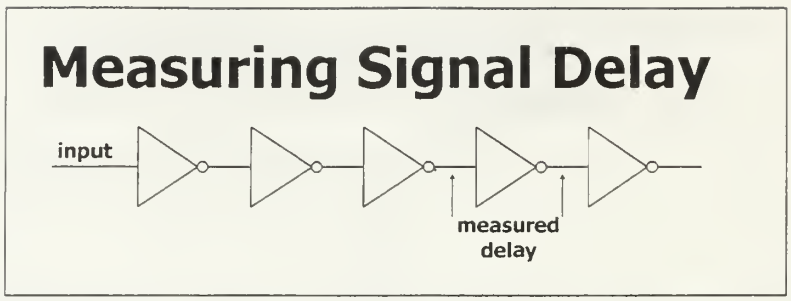


Figure 17. Delay Circuit for Measuring Inverter Response.

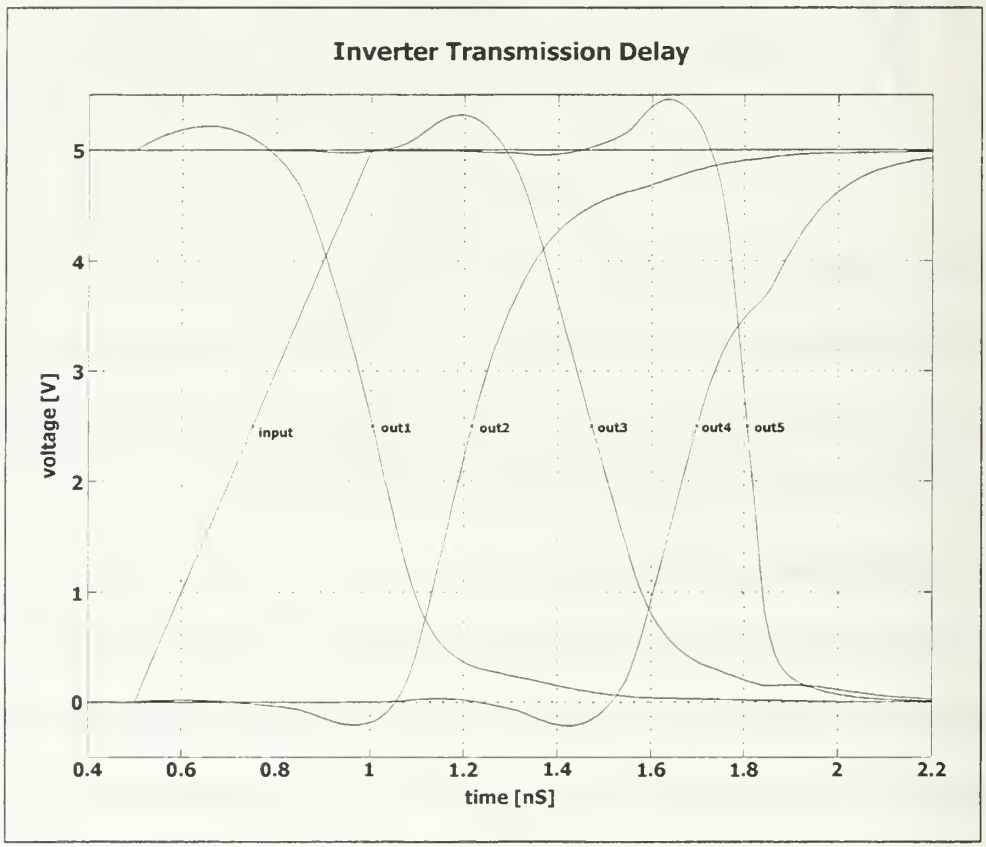


Figure 18. Inverter Transmission Response for Delay Circuit.

Output Transition	Delay
High to Low	0.2564 nS
Low to High	0.2240 nS

Table 21. Inverter Delay Summary.

B. LOGIC ELEMENT DESIGN

Using a 6 μm PFET width and 3 μm NFET width, the basic logic elements were designed to support VLSI implementation of the structural TIC design. Appendix E contains the specific design and evaluation data regarding all logic elements. Table 22 presents a summary of logic element response for each design. The delay values were obtained while simulating an output loading of 0.1 pF; equating to 4 down-stream components. When compared to the 500 nanosecond half-cycle of the 1MHz clock, even the worst transmission delays allow numerous components to be connected in a series layout while still providing adequate stabilization time.

Component	Transistors	Slowest Delay	Peak Power
Inverter	2	0.83 nS	2.1 mW
2-input NAND	4	1.33 nS	2.3 mW
3-input NAND	6	1.69 nS	2.7 mW
4-input NAND	8	2.22 nS	2.7 mW
2-input AND	6	1.31 nS	2.3 mW
2-input NOR	4	1.51 nS	1.6 mW
3-input NOR	6	2.10 nS	1.2 mW
2-input OR	6	1.47 nS	2.5 mW
2-input XOR	12	1.68 nS	3.2 mW
2-input XNOR	12	1.68 nS	3.2 mW
D flip-flop with nClear	24	3.42 nS	3.4 mW

Table 22. Component Design Summary.

C. COMBINED LOGIC COMPONENT CONSTRUCTION

The storage registers, down counters, shift registers, and other components were developed using the structural designs presented in Appendix B. The logic element layouts

were combined to create rows of elements with a common power and ground line. This configuration produced the most compact layout of the larger components. The logic elements were arranged to provide the shortest connection distance between elements and signals were routed primarily on the first metal layer or the polysilicon layer. Signals that extended beyond adjoining elements were typically routed vertically on the second metal layer, then routed horizontally on the first metal layer, and again vertically on the second metal to their destination. This routing direction segregation reduced wasted space between component rows and helped to maintain a very tight layout.

E. MODULE ASSEMBLY

The components were arranged on the VLSI layout area to minimize the distance between signal generation and signal use. Sets of signals were routed together to maintain an orderly structural layout. The internal command bus was routed between the components that accessed its values, minimized the bus size and its drive loading. The modules were built to obtain a consistent length. The modules were then attached to the common power and ground buses running vertically along both sides.

F. INPUT AND OUTPUT CONSIDERATIONS

Movement of input signals to the TIC and output signals to the current switching network required an arrangement of input and output pads. These pads are bonded to tiny wires that connect the chip to the DIP package. A power and ground ring encircle the chip just inside the connection pads. These rings provide voltage surge protection through diodes designed into the I/O pad structure. In addition to the power and ground pads that feed the outer rings and the chip components, three pad types are included in this design.

A standard input pad is used to translate the data and clock inputs onto the lines connecting them to the TIC circuit. These pads are comprised of two inverters that act as a signal buffer and amplifier.

A standard output pad is used to convert the TIC output into a signal strong enough to drive the current switches. Again, these pads are made using two inverters that are specifically sized to provide the proper amount of current to activate the switching network.

A custom input pad was developed to act an input jumper signal. The pad was held high through a diode and resistor combination, providing a logic "1" to the circuit unless the pad was jumped to ground. When this type of pad was grounded, a logic "0" was provided to the circuit. These pads were used for creating the address assignment jumpers and the frequency and reset selection jumpers.

G. COMPLETE TACTOR INTERFACE CHIP

When the pads were all combined with the entire system layout, the TIC was complete. Figure 19 shows the layout for the entire TIC. Using the layout map provided in Figure 20, the relative size and placement of each component is clearly visible.

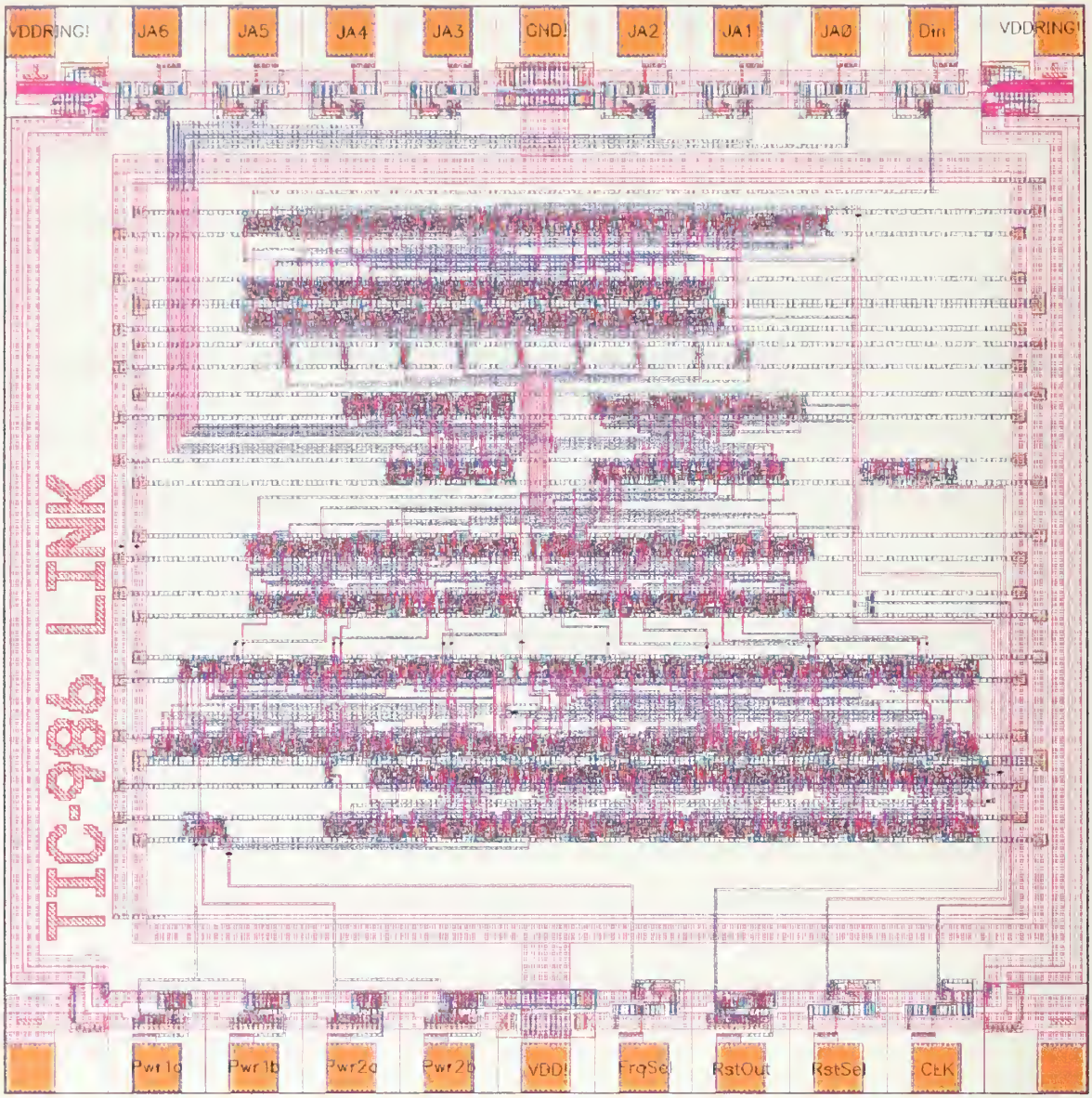


Figure 19. Completed Tactor Interface Chip VLSI Design.

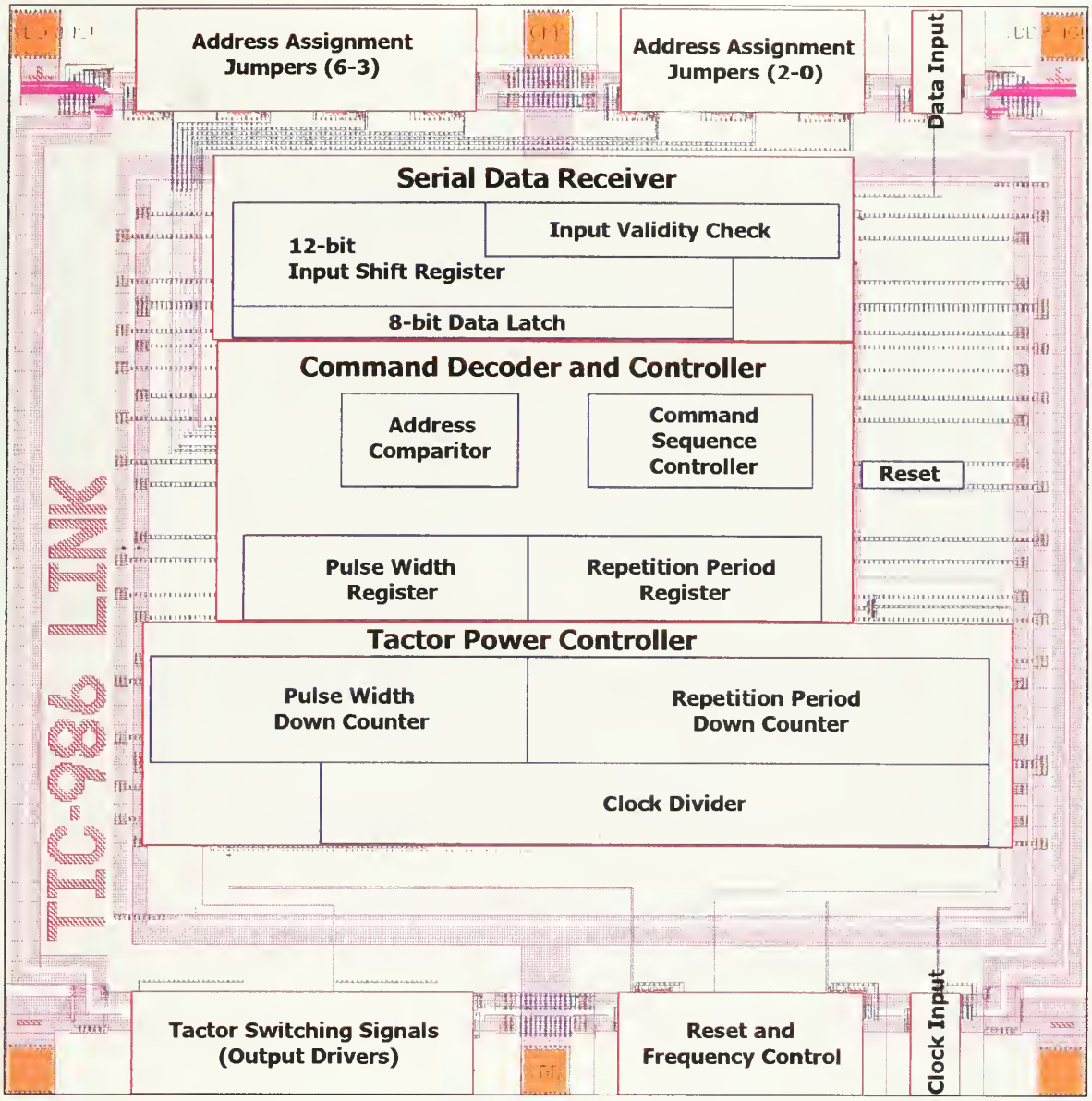


Figure 20. Layout Map of the Tactor Interface Chip VLSI Design.

H. COMPREHENSIVE SYSTEM TESTING

When the entire layout was complete, several simulations were performed on the extracted model. These simulations each ran for approximately one day and generated nearly one gigabyte of data. The results presented in Figure 21 illustrate that the TIC circuit functions as designed and produces the alternating switch control signals required to drive the tactor current network.

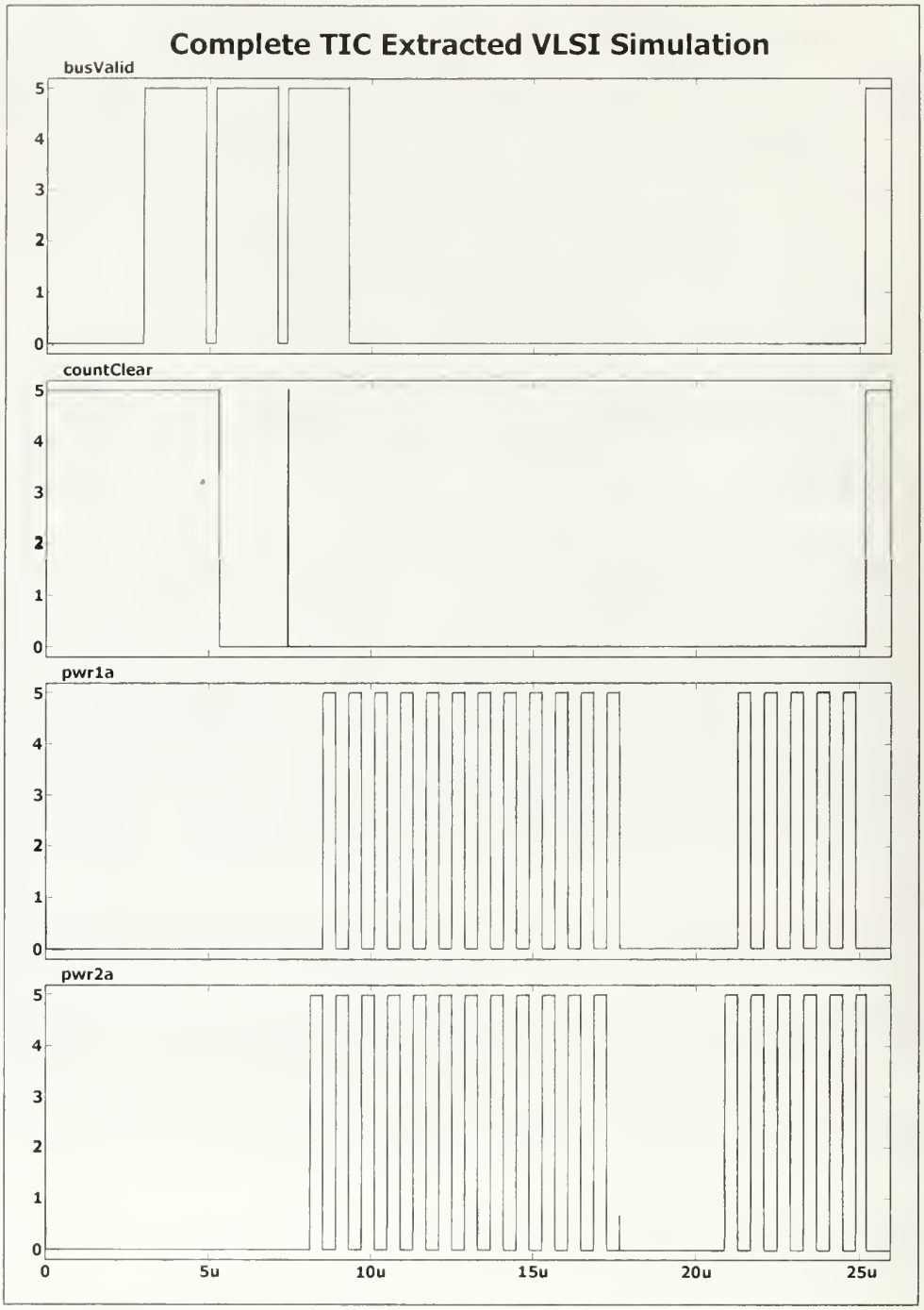


Figure 21. Simulation Results from the Complete TIC Design.

V. PARALLEL PORT DATA MODULATOR

With the VLSI layout complete and submitted for fabrication, chip testing and demonstration became the highest priority. Once fabricated, the most realistic test would be to connect the TIC to an actual serial communication line and measure the current switching signals produced as a result of issuing command bytes to the TIC. Additionally, an apparatus that could drive the TIC in a manner similar to its intended implementation would serve as a demonstration platform for the completed tactile interface.

The general notion of using a standard computer to produce the serial bit stream required for TIC operation was appealing for many reasons. The primary motivation for PC use was the portability of such a system; system design could allow using any PC, reducing system unique equipment to a small hardware component and the tactile array. Additionally, the parallel port on a computer closely represents the operation of a microprocessor data bus. This similarity to a data bus provides a level of design abstraction that would ease adaptation of the command modulator to work with any micro-controller.

This chapter presents many of the aspects in the development, fabrication, and testing of the Parallel Port Data Modulator. After introducing the conceptual design, parallel port transfer characteristics and modulator design specifics are discussed. Next, the circuit board layout and manufacture for the modulator are included. The command software driver is introduced and, finally, system testing and modifications are discussed.

A. CREATING A SERIAL COMMAND STREAM

A flexible and effective method to issue commands to the intelligent tactor is to write a computer program that presents the byte command to a modulator attached to the computer parallel port. The modulator, shown conceptually in Figure 22, reads the byte

presented at the parallel port and latches it into a transmission buffer. The hardware interface then signals the computer that the command has been read to allow computer processing for the next command byte. The modulator transmits the command using the required serial data packet format. TIC synchronization and timing is provided using the 1 MHz clock.

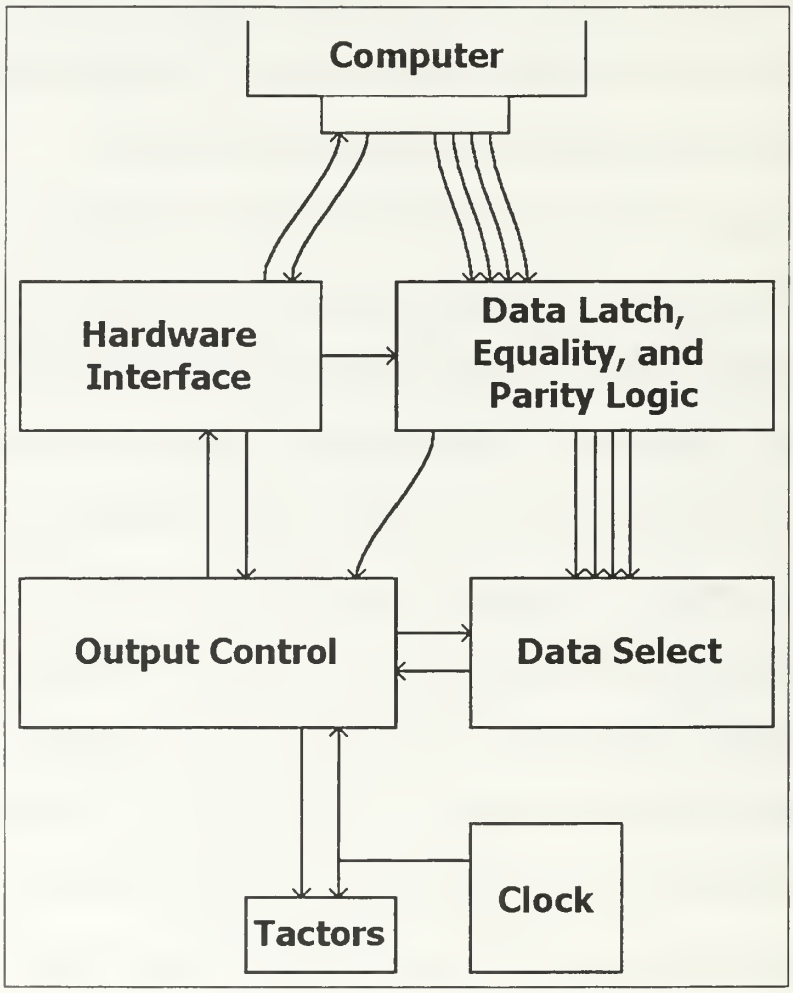


Figure 22. Command Modulator Conceptual Design.

B. PARALLEL PORT INTERFACING AND CONTROL

The parallel port on a computer is used to communicate data one byte at a time to an attached peripheral. Reference 1 contains an extensive description of parallel port

communications. The transfer of data through the parallel port is defined by IEEE standard 1284, "Standard Signaling Method for a Bi-directional Parallel Peripheral Interface for Personal Computers." The parallel port consists of 17 signal lines that are divided into three categories: data (8 lines), control (4 lines), and status (5 lines). The remaining eight lines are ground connections. Table 23 summarizes the signals with their descriptions and connector pin assignments. Figure 23 illustrates the parallel port connector and pin numbering scheme as viewed on the back of a computer.

Category	Name	Pin	Direction	Description
Data	data 0	2	In/Out	Active high. Data transmission lines. Operate only in output direction for some communication modes.
	data 1	3		
	data 2	4		
	data 3	5		
	data 4	6		
	data 5	7		
	data 6	8		
	data 7	9		
Control	nStrobe	1	Out	Active low. Indicates valid data is on the data lines.
	nAutoLF	14		Active low. Instructs printer to automatically insert a line feed for each carriage return.
	nInit	16		Active low. Resets device.
	nSelectIn	17		Low signals device it is selected.
Status	nError	15	In	Low indicates an error exists.
	Select	13		High indicates device is online.
	PaperEnd	12		High indicates printer is out of paper.
	nAck	10		Low indicates last byte was received.
	Busy	11		High indicates device is busy.

Table 23. Standard Parallel Port Signal Definitions and Pin Assignments.

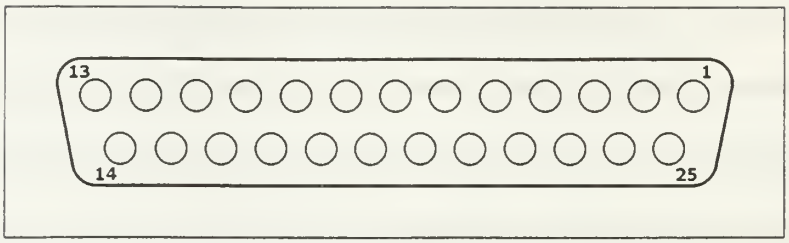


Figure 23. Parallel Port Connector with Pins Numbered.

Parallel port data transfer follows a specific procedure for every byte sent to the attached peripheral. First, the port status is checked to ensure that the peripheral is not busy and no errors are present. The data byte is then placed onto the data pins and the strobe is activated to indicate that the data on the data lines is valid. The strobe is held active until an acknowledgement is received from the peripheral that indicates data receipt. This process continues until all bytes have been successfully transmitted.

C. MODULATOR DESIGN SPECIFICS

The hardware interface of the command modulator uses the eight data lines, strobe signal, acknowledge signal, and busy status. When the computer presents a byte to the modulator, it issues a data strobe. The modulator detects the strobe and activates the busy flag to prevent other bytes from being transmitted. Then, the command is latched into the buffer and, when the buffer value matches the input command, an acknowledgement signal is sent to the computer. The modulator then cycles through the data packet format using an internal state machine. Serial bits are placed on the TIC communication bus on the negative clock transition to ensure they are stable when latched at the TIC on the positive clock cycle. The start bit is transmitted first, followed by the command byte proceeding from the most significant bit to the least significant bit. The parity bit and stop bit are then sent, completing the cycle. After the stop bit is sent, the busy signal is cleared to allow the next byte to be latched into the modulator buffer.

The logic functions for the command modulator were created using Programmable Logic Devices. PEEL 18CV8P PLDs were used to implement the logic functions because they were available in the research laboratory. The choice to use these chips directly influenced the method of implementing the conceptual design shown in Figure 22. The design requires the buffer to produce an equality function and parity calculation. The equality function triggers the acknowledgment signal that is fed back to the computer. Unfortunately, the 18CV8P chips have insufficient logic capacity to perform all of these functions on a single chip. In fact, to create a discrete buffer, equality, and parity functions requires three individual chips. However, these functions can be realized with two PLDs by creating a 4-bit latch that includes partial equality and parity calculations. Using equality and parity inputs, the two cascaded chips will perform all three functions.

A more detailed discussion of the Parallel Port Data Modulator design is included in Appendix F. The appendix contains the Verilog® modeling source files and the ABEL™ logic definitions used to create the required JEDEC format data files for PLD programming.

D. CREATING A PRINTED CIRCUIT BOARD LAYOUT

The Parallel Port Data Modulator physical characteristics were generally defined by the system goals. The command modulator needed to be a small, self-contained device that connected directly to a computer parallel port thus compact board size was a high priority. The basic component layout was conceptualized as the PLD programs were being developed. Figure 24 illustrates the layout for the command modulator components. The system consists of a parallel port connector, four 20-pin DIP sockets, one 8-pin DIP socket, a wiring harness connector, and power connections. The complete system measures about $2^1/_2$ inches by $2^1/_2$ inches and uses a battery pack of four AA batteries.

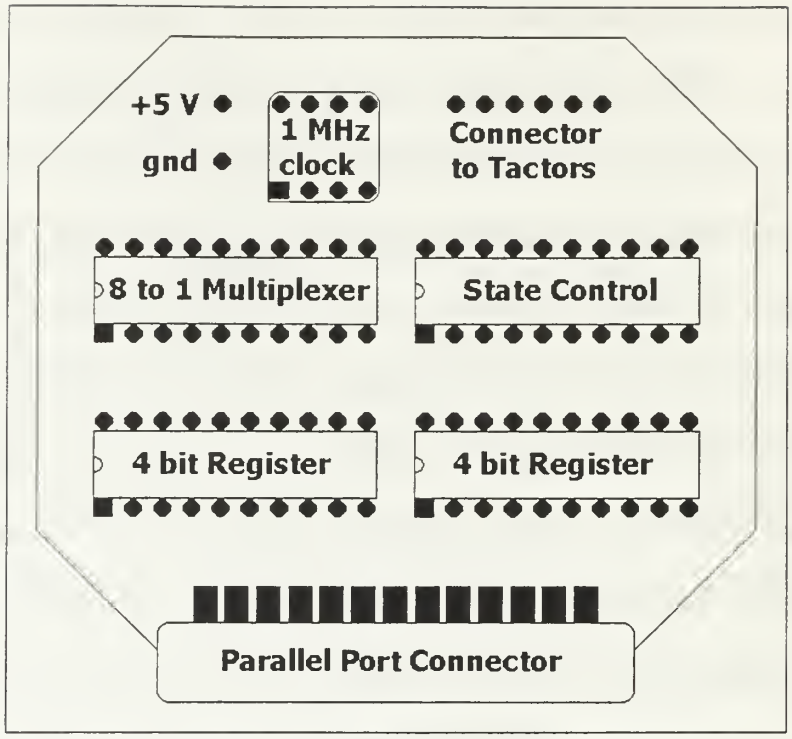


Figure 24. Parallel Port Modulator Component Layout.

The PLD logic programs were developed first to ensure the required functions would fit on four chips. After the chips performed as required, adjustments to the chip pin assignments and design were made to support the desired circuit board layout and to simplify signal routing on the board. The signal and power lines were then routed on each layer of the two-layer board. Figure 25 and Figure 26 show the layout and routing of the manufactured circuit board for the top and bottom layers, respectively. For consistency, power and ground were routed exclusively on the bottom layer and data signals were routed primarily on the top layer. In hindsight, a better layout plan would have considered the solder connections for each component to determine which layer would be best for the signal to reach the pad. Soldering was typically easier to perform for bottom-layer signal pads because the wiring harness connector and DIP sockets were mounted to the top layer.

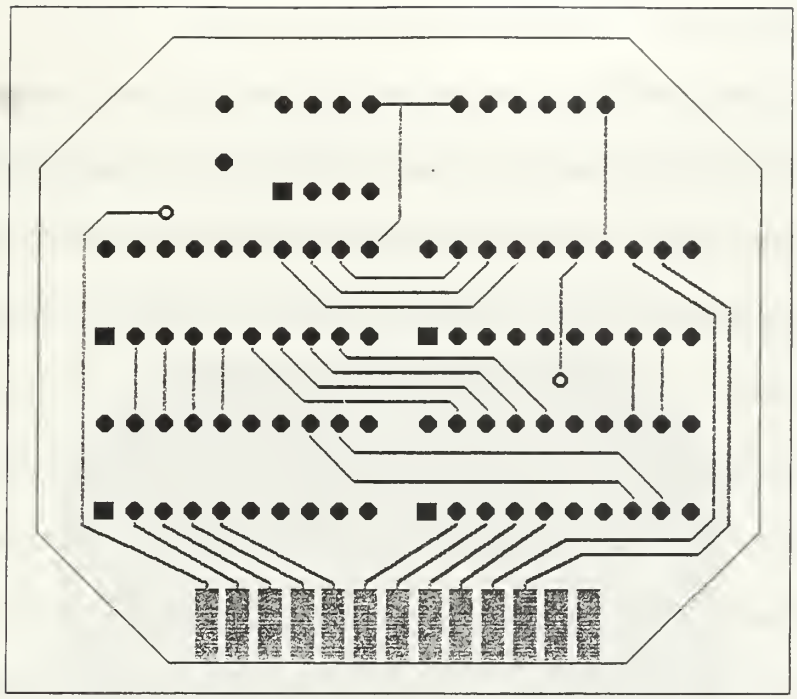


Figure 25. Parallel Port Modulator Top Layer Routing.

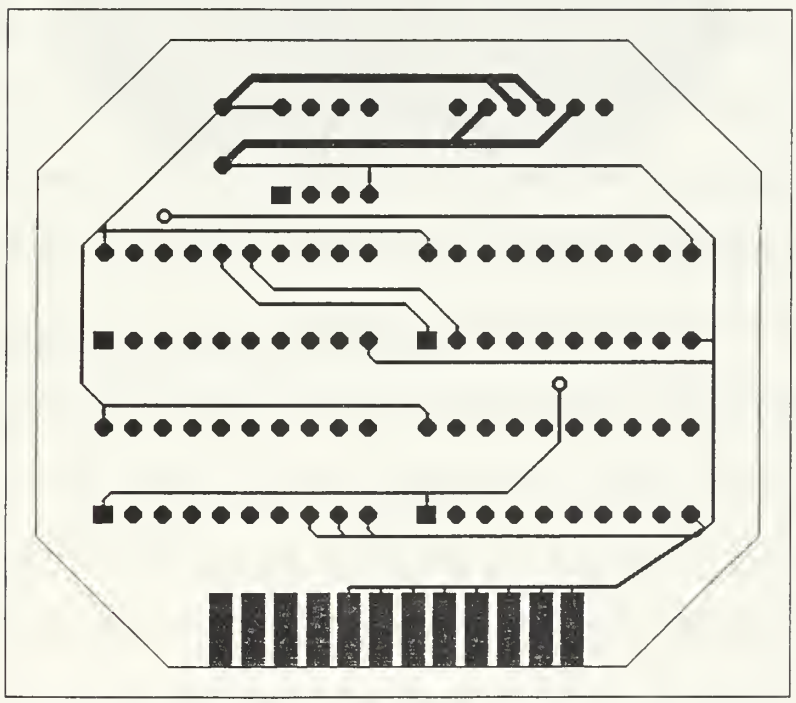


Figure 26. Parallel Port Modulator Bottom Layer Routing.

E. PRINTED CIRCUIT BOARD MANUFACTURING

Prototype printed circuit boards can be manufactured from a copper-coated insulator board by removing the copper from areas around the desired conductors. Two processes exist for copper removal: chemical etching and machining. The chemical removal process uses a resistive mask to protect the desired conductor areas while immersing the board in a chemical to remove the copper in the unprotected areas. The machining process uses a digitally controlled milling machine to mechanically remove the copper around the conduction paths to isolate the conductor from the remainder of the board. The chemical process was used in the first attempts to manufacture the Parallel Port Data Modulator. The rather crude masking methods used in the process produced marginal results. The milling process was then used with much greater success.

To produce a circuit board using the machining process, a GERBER data file containing the layout and routing data is necessary. The GERBER file is produced using circuit board layout software. EasyTrax (ver 2.06) by Protel International Pty. Ltd. was used to layout and route the command modulator. When all layout and routing was complete, EasyTrax was used to produced GERBER output files for the top and bottom layers.

A digital milling machine made by LPKF CAD/CAM Systems, Inc. was used to produce the circuit board. This milling machine is designed specifically for making prototype boards. The CAD/CAM package includes IsoCAM software that calculates isolation channels from the GERBER files and drives the milling machine when design processing is complete. When manufacturing the board, IsoCAM first prompts the user to install a drill bit into the milling spindle to bore the holes for component mounting. Next, the appropriate cutter must be installed to produce insulation gaps around all conductors.

After the top layer is machined, the board must be flipped and aligned in preparation for machining the bottom layer. Figure 27 shows the top layer of the circuit board produced using the machining process. Note that the machining process does not remove the copper in the unused areas of the board unless specifically required.

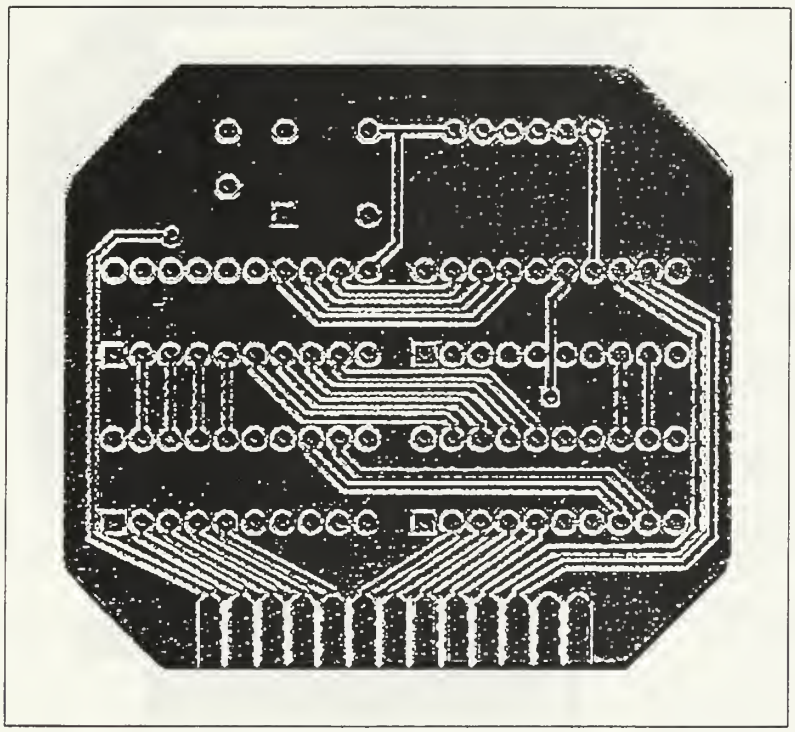


Figure 27. Command Modulator Top Layer after Machining.

F. PRINTED CIRCUIT BOARD ASSEMBLY

After fabricating two copies of the circuit board, the connectors and DIP sockets were soldered onto each board. It was at this point that the cost of not routing signals to the bottom contact pads was fully realized. The parallel port connector was easy to attach since the board thickness matches the spacing between the connector solder lugs. By slightly elevating each DIP socket, soldering the top-layer connection pads was made easier. On the other hand, the plastic edges on the connector for the tactor array wiring harness had to be carved to allow access to the soldering pads. Once the solder connections were made,

command modulator assembly was completed by placing the PLDs and crystal oscillator into their sockets. Figure 28 shows a completed Parallel Port Data Modulator after full assembly.

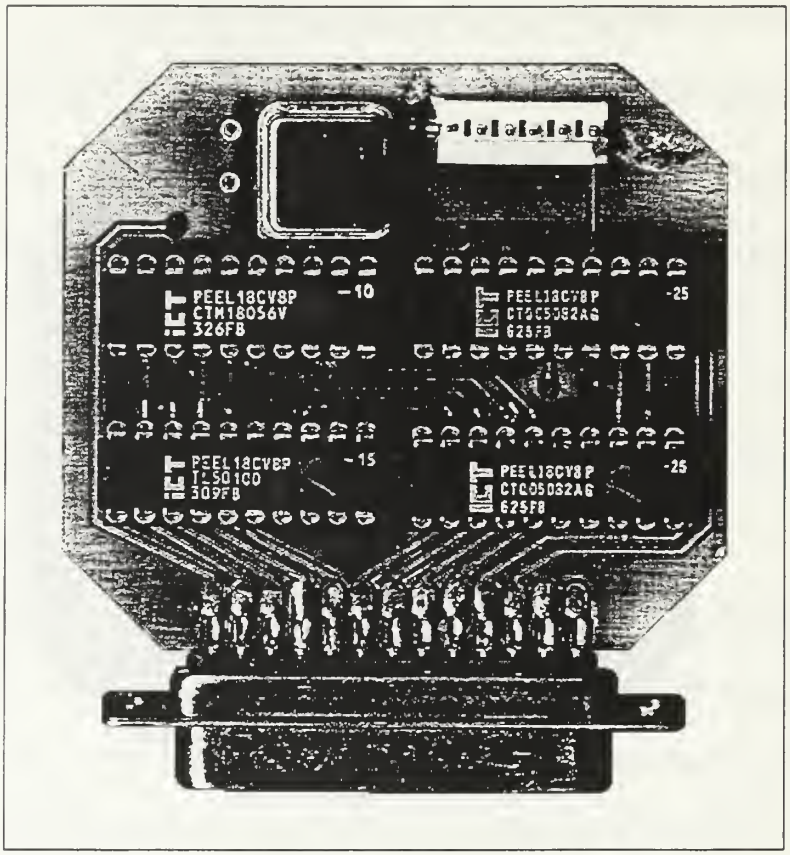


Figure 28. Fully Assembled Command Modulator.

G. SOFTWARE TO DRIVE THE COMMAND MODULATOR

For proper operation, the host computer must provide the command bytes to the modulator. The C++ programming language was used to write a driver program to facilitate issuing commands to the modulator through the parallel port. The program requests an input command and waits for a user response. When a response is detected, the program places the byte on the parallel port and waits for peripheral acknowledgement. Once acknowledgement is received, the program requests another command from the user. The

program was written for the DOS operating system to allow use with older computers. A more extensive command transmission program description and C++ source code are included in Appendix G.

H. COMMAND MODULATOR TESTING

After manufacture and assembly, the command modulator was tested to ensure proper operation. With an oscilloscope connected to the clock and data lines of the wiring harness, the command transmission software was used to issue commands to the tactile array. The oscilloscope measured the output waveforms. Figure 29 and Figure 30 show the oscilloscope displays after issuing a 19 and a 218 command respectively. The images show that the command byte is transmitted in the required serial packet format with the communication bus changing value on the negative clock transition. The apparent instability of the clock pulses is actually being caused by a noisy data probe.

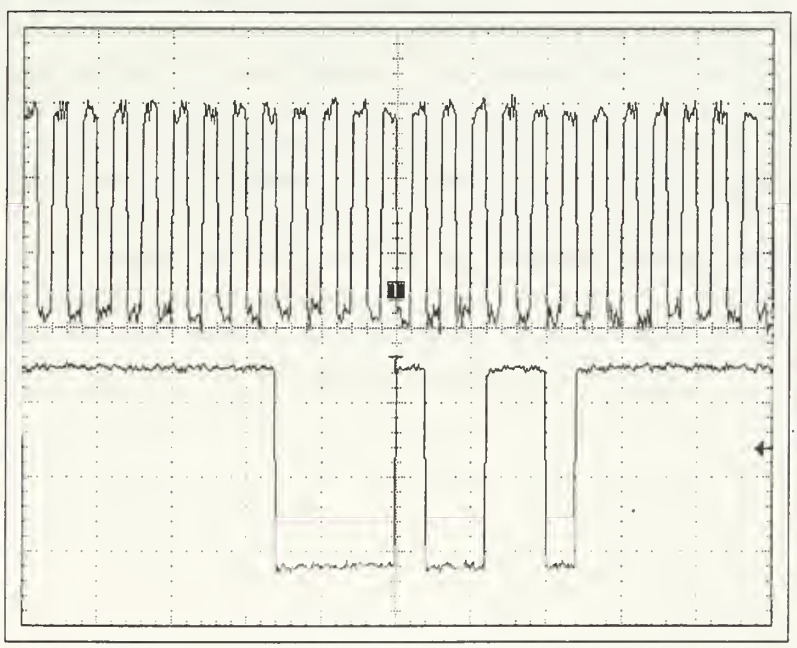


Figure 29. Command Modulator Output for 19 Command.

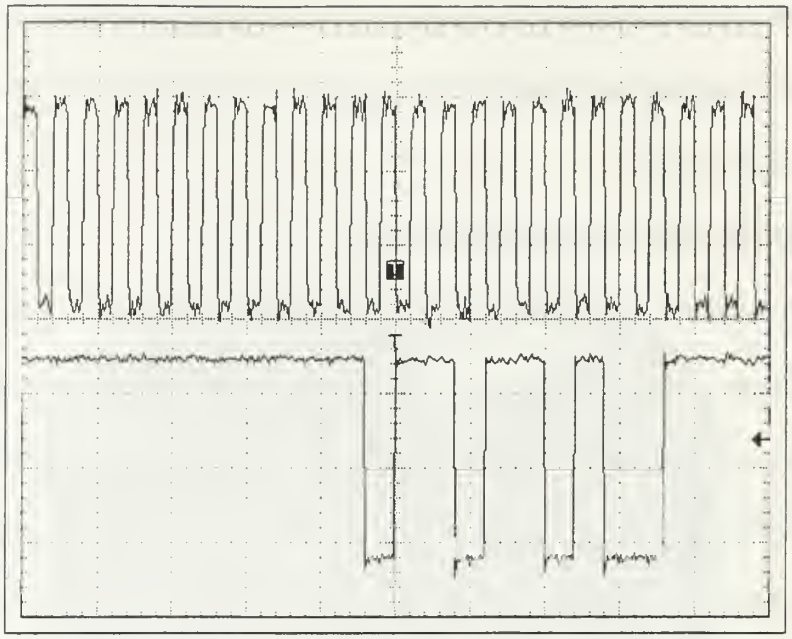


Figure 30. Command Modulator Output for 218 Command.

I. MODIFICATIONS TO THE MODULATOR DESIGN

During initial testing of the command modulator, it was discovered that several copies of the command packet were being transmitted for each ordered command. This system response was a result of the parallel port speed for the computer being used to issue commands to the modulator. The modulator was latching, acknowledging, and transmitting the command before the computer was able to clear the strobe. This anomaly required reprogramming the PLDs to delay command transmission until the computer cleared the strobe signal.

VI. TACTOR INTERFACE CHIP TESTING

Once the Parallel Port Data Modulator was complete, chip testing should have centered on wiring the TIC to receive power and commands from the modulator. When the four chips were received from fabrication, they were first inspected using a microscope to examine the general chip condition and ensure the provided pin assignments were accurate. The chips were then operationally tested using the command modulator but no output was produced. In order to identify the reason for improper operation, the complete circuit was simulated again. When the design simulated properly, the chips were more closely examined using a scanning electron microscope and some potential manufacturing problems were identified. Charged electron imaging was then attempted without success. Further testing is not planned for this chip.

A. VLSI CHIP RECEIPT FROM FABRICATION

Four copies of the TIC were fabricated and bonded into 28-pin DIP packages. The chips were mounted in anti-static foam and protected by a hinged plastic box. No damage was evident to the chips due to the packaging and shipping processes. The TICs came with a data sheet indicating the pin assignments that resulted from packaging. Figure 31 represents the TIC schematic with the signals associated to each pin.

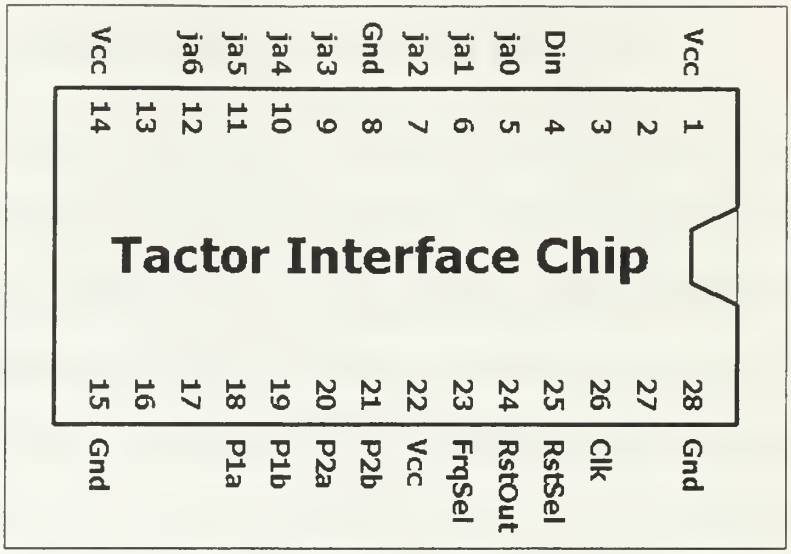


Figure 31. Tactor Interface Chip Pin Assignments.

B. VISUAL INSPECTION

The four chips received from fabrication were inspected using a microscope to examine the general chip condition and ensure the provided pin assignments were accurate. The stated pin assignments were correct, but microscopic inspection of the chips revealed several dark areas that were initially thought to be dust on the protective scratch coat. A more detailed visual inspection subsequently indicated that some of the impurities are in the same fabrication layer as the aluminum conductors and may even extend into the silicon transistor areas.

C. OPERATIONAL CHECK USING COMMAND MODULATOR

After confirming pin assignments, a wire wrap test circuit was constructed to mate the TIC to the command modulator. All connections were traced and verified prior to energizing this single-element tactile array. When power was applied, the clock signal was measured at the TIC and found to be correct. The serial bus was then monitored as command bytes were transmitted to the intelligent tactor. Each command packet was

received as expected but the TIC failed to provide the anticipated response. After confirming the system setup and verifying all signal paths, several more commands sets were issued. Still, no response was obtained from the mounted TIC. A second TIC was mounted to determine if the first chip was faulty. The same series of tests were performed and there was no response from the second TIC.

D. COMPLETE SYSTEM RESIMULATION

One likely cause of circuit failure could have been a faulty design. Rather than continue with operational testing a plan was made to completely verify the TIC circuit again to determine if a design oversight had been missed in the original testing. All original tests were performed again on the extracted VLSI design. Every aspect of the circuit responded exactly as designed. Additional tests were conducted to precisely simulate the series of commands used to operationally test the TIC. Again, the simulation responded exactly as specified.

E. SCANNING ELECTRON MICROSCOPE INSPECTION

Access to a scanning electron microscope was obtained to investigate the TIC response failure. During careful examination of all chips, several manufacturing problems were detected on every chip. The left image of Figure 32 shows contamination that may be causing a short between power and ground. Spectral analysis of this area indicated that the contaminant contained high levels of sulfur. The right image of Figure 32 shows particulate contamination that might be shorting between the signals on the TIC internal command bus. Figure 33 shows areas of aluminum oxidation. Figure 34 shows a metalization failure in the top aluminum interconnect layer that causes the metal to extend beyond its design channel. Finally, Figure 35 shows some of the many impurities peppered throughout the entire chip

layout. While these manufacturing problems may not be the direct cause of chip failure, they certainly indicate questionable fabrication cleanliness. The real concern is not the areas that were examined using the scanning electron microscope. The microscopic examination only shows problems in the visible layers at the top of the silicon wafer. If these images reflect general fabrication quality, the most likely cause of chip inoperability is similar impurities and failures in the lower fabrication layers.

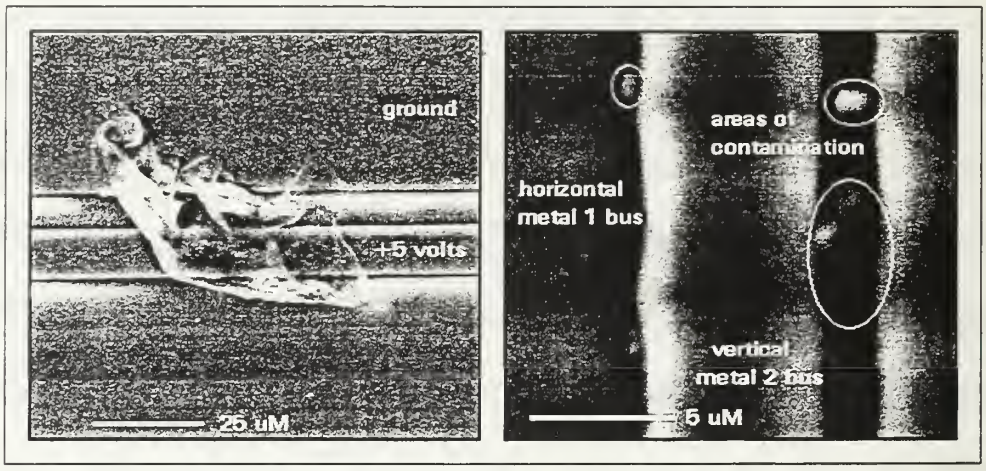


Figure 32. Scanning Electron Microscope Images of Potential Shorts.

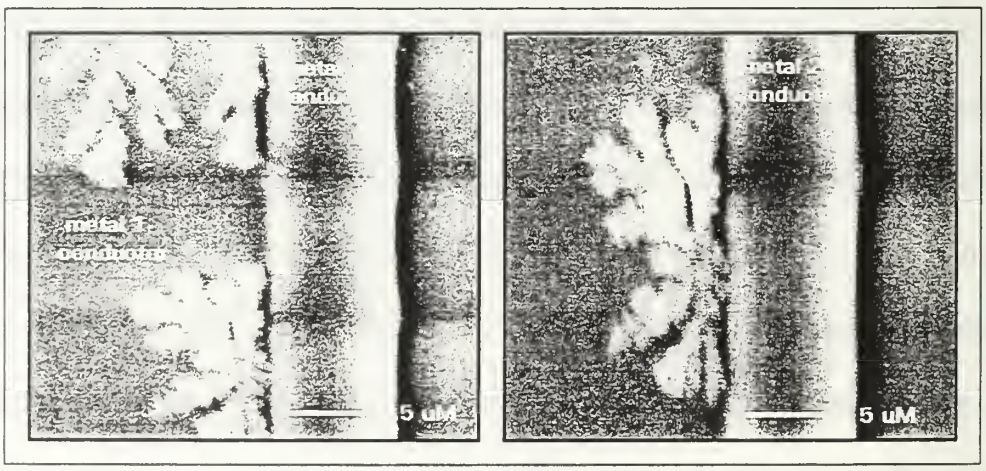


Figure 33. Scanning Electron Microscope Images of Aluminum Oxidation.

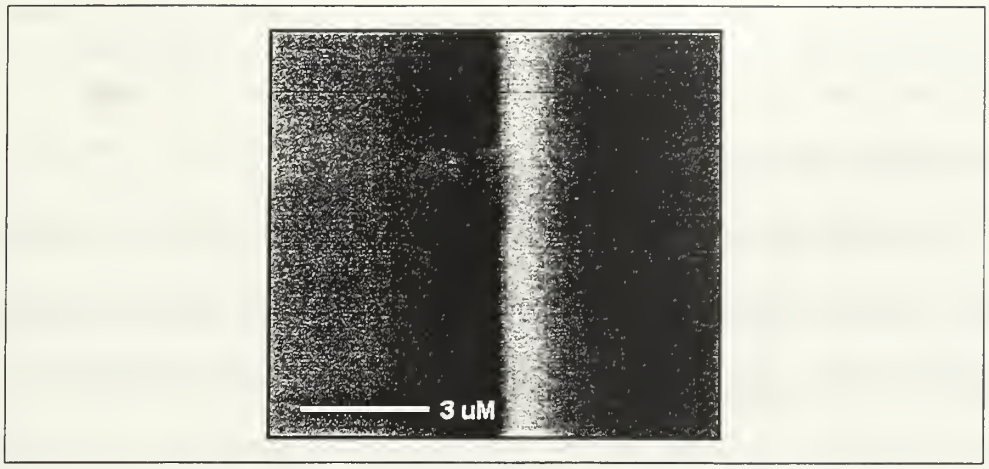


Figure 34. Scanning Electron Microscope Image of Mask Failure.

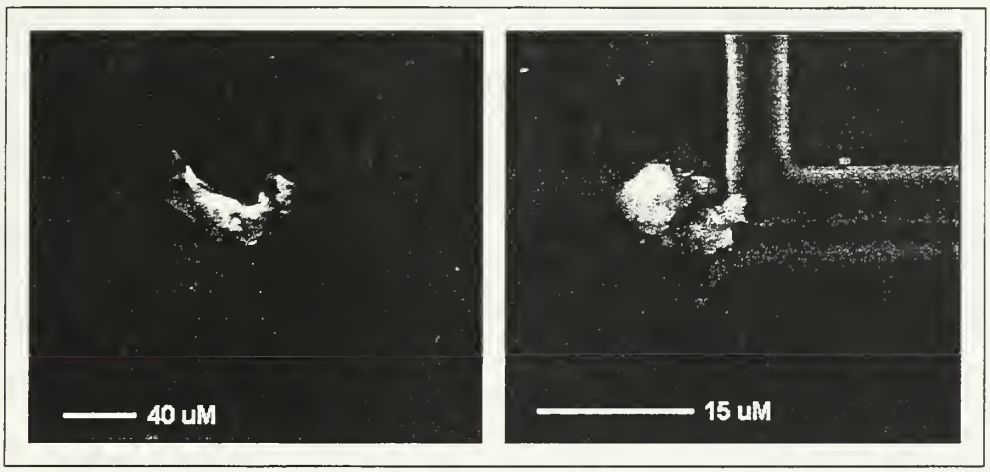


Figure 35. Scanning Electron Microscope Images of Embedded Impurities.

F. CHARGED ELECTRON IMAGING

Charged electron imaging is a method for observing the microchip using a scanning electron microscope while the chip is energized to determine operating conditions. The areas of the chip that are at a higher potential appear much brighter than the areas that are grounded. This examination provides a visual method for circuit analysis with respect to operations and points of failure. A special circuit was built to clock one bit per second into the TIC to support charged electron imaging. When the chip was tested using this method, there was no visible contrast between the power and ground points. This indicates the

inspection procedure was more complex than initially understood. Further investigation by the microscope technician is in progress to support this testing in the future.

G. FURTHER TESTING

Very little further testing is expected since the TIC chip is not currently funded research. The next generation chip should be constructed with various test points to allow evaluation of circuit performance at different locations within the VLSI layout.

VII. REVISIONS TO THE COMMUNICATION PROTOCOL

The basic command structure is very efficient for communicating the essential information for the tactile interface. Now that the first iteration is complete, the command structure must be reevaluated for improvement with a second-generation tactile interface. Many additional instructions could be included in the basic TIC control language. A redefinition of the command structure will also require significant changes to the TIC design. This chapter presents the limitations of the current command structure and suggests a revised command structure that will improve system response and flexibility.

A. EVALUATION OF REGISTER COMMAND PAIRS

The current command structure produces continuous tactor activation for any wave-shape parameter pair that has a pulse width greater than or equal to four times the repetition period. The duplication of response for different register values is an area available for command improvement.

A constant resolution of 16 mS for pulse width and 64 mS for repetition period is easy to implement with the first-generation down counter scheme. However, this timing method produces an extremely wide difference for percentage wave-shape resolution. When operating with the maximum repetition period, the wave shape has 64 different selections for duty cycle between 0 and 25 percent and a duty cycle resolution of 0.4 percent. However, when operating with a one-half second repetition period, the wave shape can assume 32 different duty cycles ranging from 0 to 100 percent with a duty cycle resolution of 3.1 percent. As repetition period continues to decrease, the duty cycle resolution increases exponentially. A more consistent duty cycle resolution would better represent the desired physical stimulation.

A command structure based on duty cycle rather than pulse width would improve both concerns in the preceding paragraphs. Duplicate response to command pairs would still occur once for 100 percent activation at each value of repetition period but all other duplication would be limited to the minimum resolution for the controlling counter. A duty cycle parameter would also define a consistent duty cycle resolution at all repetition period values. Use of 16 discrete duty cycles would provide a consistent 6.25 percent resolution while 32 duty cycle values would provide 3.125 percent resolution.

B. TACTILE ARRAY SIZING

The original target tactile interface included forty tactors. The concept of using multiple address values for each tactor was considered as a viable method of improving system response by defining group identifiers in addition to the unique individual address. Subsequent consideration of potential tactile interface applications supports forty tactors as nearly the maximum number possible rather than an initial estimate. At the 1 MHz serial transfer speed, the need for group addressing schemes is not critical since thirty three-byte command sets can be issued in less than 1 mS. From these two assertions, the choice of 126 individual tactor addresses is too high and consumes too many of the 256 available commands. Use of 63 individual addresses and one "all call" is sufficient for all expected tactile interface applications.

C. PROGRAMMABLE OSCILLATION FREQUENCY

During the course of this research, the expected tactile transmitter has been changed three times. Each new tactor operates best at a specific oscillation frequency and drive current. The range of operating frequencies has been from 100 Hz to 250 Hz. Although the current TIC design supports two discrete frequencies, a TIC capable of altering the

oscillation frequency using a command would be much more flexible for evaluating an array of currently undefined tactors.

D. COMMANDED RESET

In certain circumstances, it might be beneficial to force a system reset for an individual or group of tactors. A dedicated reset command allows an explicit reset to be executed by any tactor interface chip.

E. REVISED COMMANDED STRUCTURE

The command structure defined in Table 24 below balances the concerns in the preceding sections with the 256 available byte commands.

Command Word	New Meaning
0 0 0 0 0 0 0 0	Reserved -- TIC bus idle condition
0 0 0 0 0 0 0 1	Explicit System Reset.
0 0 0 0 0 0 1 0 to 0 0 0 0 1 1 1 1	Unused commands available for future use (14 values).
0 0 0 1 0 0 0 0 to 0 0 0 1 1 1 1 1	Oscillation Frequency (16 discrete values).
0 0 1 0 0 0 0 0 to 0 0 1 1 1 1 1 1	Duty Cycle (32 values ranging from 3% to 100%).
0 1 0 0 0 0 0 0 to 0 1 1 1 1 1 1 0	Addresses for up to 63 tactors.
0 1 1 1 1 1 1 1	ALL CALL -- all tactors respond
1 0 0 0 0 0 0 0 to 1 1 1 1 1 1 1 1	Repetition Period value 0 to 127 with 32 mS resolution (0 to ~4 seconds).

Table 24. Revised Command Structure.

VIII. INCORPORATION OF ADDITIONAL DESIGN FEATURES

The initial intelligent tactor design was valuable to prove the concept is possible. The second-generation tactile interface incorporates the revised command structure and includes a design change to reduce current switching noise on the power line. This chapter presents three improvements to the basic TIC design that evolved from these two issues.

A. IMPROVED BI-DIRECTIONAL CURRENT SWITCHING SCHEME

The initial method for generating bi-directional tactor current alternately activates the diagonal switch pairs in the current switching structure of Figure 36. The implemented switching pattern is illustrated in Figure 37. The drawback to this initial switching pattern results from the switching characteristics of the bi-directional junction transistors. For a brief period, both switches on each leg are conducting, resulting in a low resistance path between the power line and ground. This momentary shorting action produces noticeable transients on the power line that may affect TIC operation. A better switching pattern is illustrated in Figure 38. The revised switching scheme prevents any shorting action on either leg of the current switching structure, greatly reducing the switching transients on the power line.

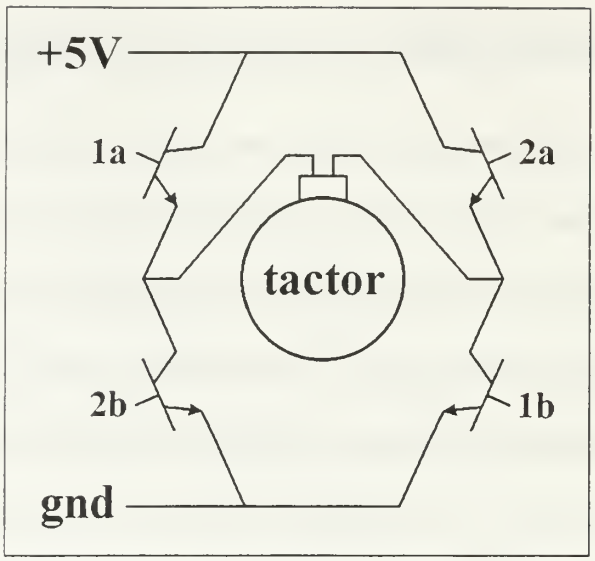


Figure 36. Tactor Current-Switching Structure.

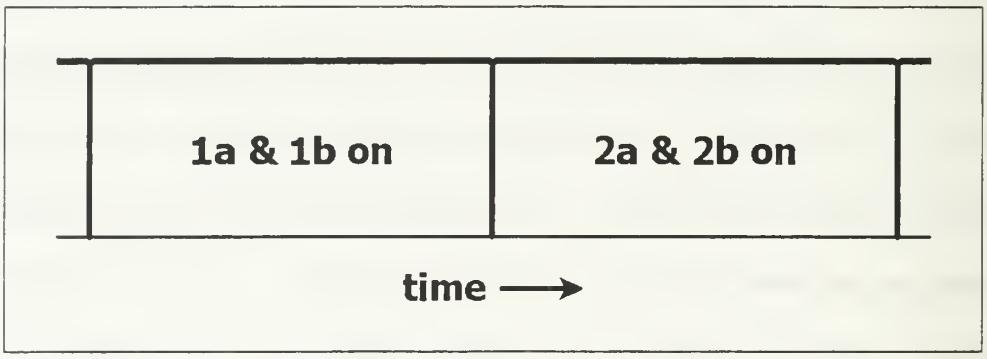


Figure 37. Initial Current Switching Pattern.

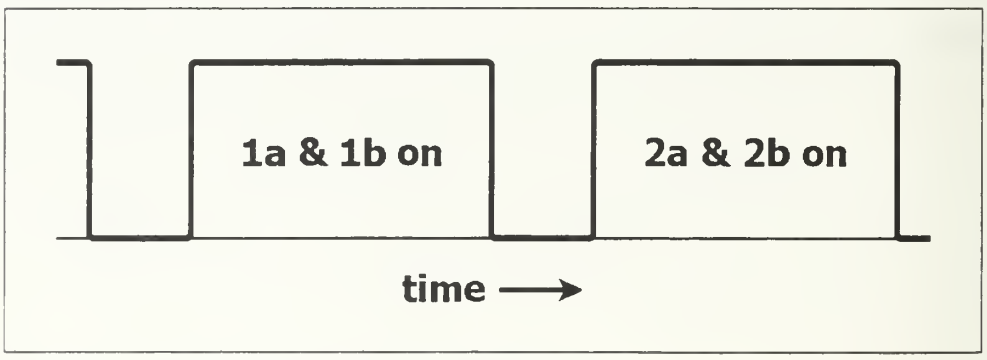


Figure 38. Revised Current Switching Pattern.

B. PROGRAMMABLE OSCILLATION FREQUENCY

Incorporating a frequency register would allow the TIC to vary the tactor oscillation frequency on the fly. Implementation of a separate frequency command is illustrated in Figure 39. The redesigned module also includes the revised current switching pattern discussed in the preceding section. To support the revised current switching pattern, the oscillation frequency generator must produce a frequency eight times the desired oscillation rate. This higher frequency then drives a loop counter whose value defines the switching pattern.

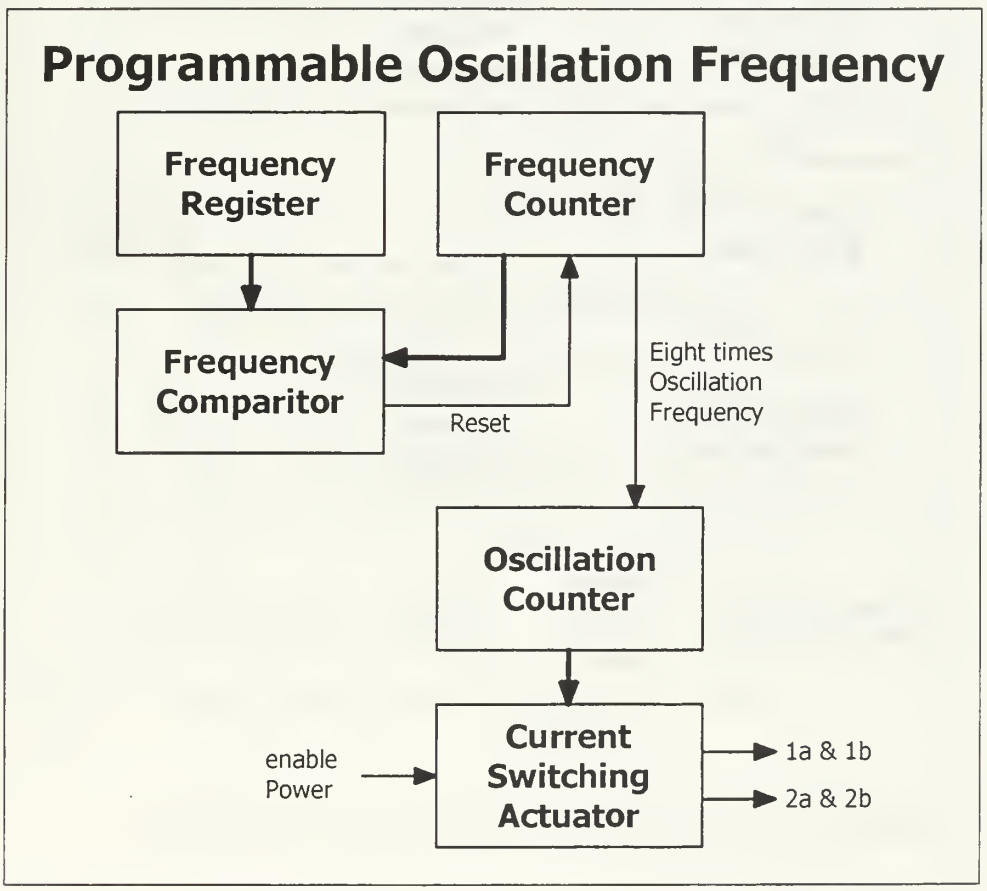


Figure 39. Generating the Oscillation Frequency with Revised Switching.

C. WAVE SHAPE GENERATION USING DUTY CYCLE

Incorporation of a duty cycle parameter instead of the pulse width parameter requires the system to calculate pulse width from the stored duty cycle and repetition period. A possible design that uses duty cycle to create the desired wave shape is illustrated in Figure 40. This design uses a single up counter whose value is compared to the stored repetition period and calculated pulse width to control tactor activation.

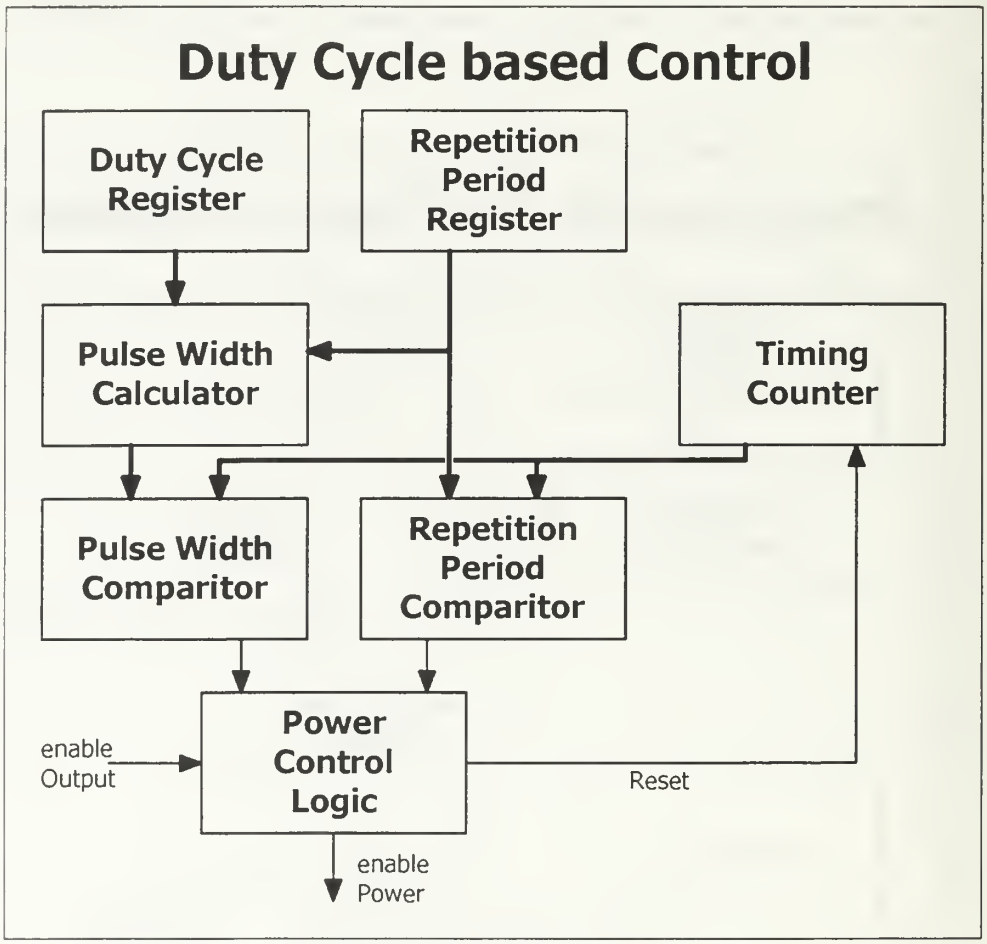


Figure 40. Wave Shape Generation using a Duty Cycle Register.

D. REVISED COMMAND DECODER AND CONTROLLER

Changes to the command structure directly affect the design of the Command Decoder and Controller module. Many changes are required in this module since the revised commands differ significantly from the original command structure. The most sweeping changes are required in the control signals produced by the command sequence controller. Figure 41 illustrates the design changes required in the Command Decoder and Controller module.

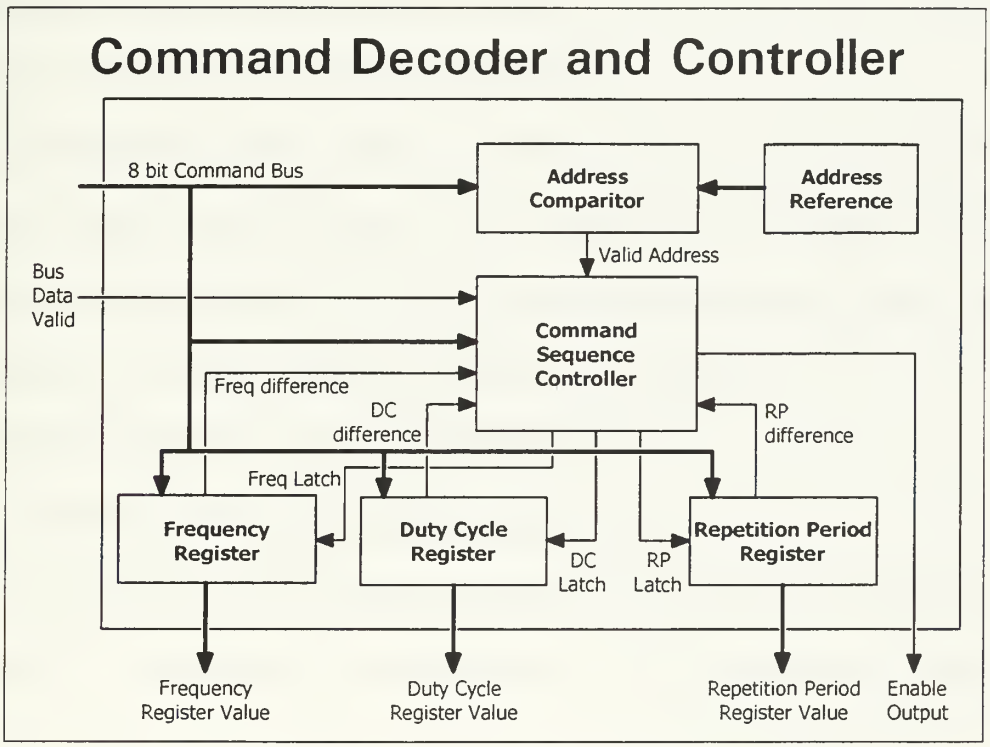


Figure 41. Revised Command Decoder and Controller module.

IX. CONCLUSIONS AND FURTHER WORK

Tactile communication is a viable method of conveying information without impeding other sensory inputs. In many applications, tactile messages may be most appropriate due to their intuitive and covert nature.

Previously, tactile communication has been experimental and limited, lacking methods to effectively implement the technology in the field. This thesis has resulted in a communication protocol and a tactor interface chip that will advance tactile communication beyond its current physiological research environment.

Implementation of this concept is currently awaiting successful VLSI fabrication. As more funding becomes available, many improvements are planned for the next generation of Tactor Interface Chips. The Naval Postgraduate School is ready to advance this technology for military, industrial, and consumer applications.

A. TACTILE INTERFACE SYSTEM PERFORMANCE

1. Simulation Performance during Design Process

Using minimum sized transistors, the tactile system has been completely designed and simulated. The simulations operate properly at all development stages using clock speeds of 5 MHz.

2. Parallel Port Data Modulator Performance

An interface that allows driving the tactile array from any commercial computer has been developed to support TIC testing and demonstration. The modulator is provided command bytes from the parallel port of a computer. The command modulator automatically interfaces with the computer to receive the data then it transmits the data in

the require serial packet format. The Parallel Port Data Modulator has been manufactured and successfully tested using simple programmable logic devices.

3. Manufactured TIC Performance

When received from the fabrication and packaging process, the TIC did not operate as designed. In fact, the TIC produced on response at all. The entire design was simulated again and found to work exactly as specified. Inspection of the VLSI chips using a scanning electron microscope revealed many questionable manufacturing issues primarily regarding cleanliness.

B. IMPROVEMENTS THAT ARE READY TO INCORPORATE

As the current chip was being fabricated, design of the next generation tactile interface began. This new design improves the original design in several ways.

1. Expanded Communication Protocol

After careful evaluation of the original communication protocol, some basic changes were made to make better use of the available command structure. The number of addresses was reduced to allow improvement in the repetition period resolution. Pulse width was discarded in favor of a duty cycle definition. Reset and oscillation frequency commands were also added.

2. Shaped Oscillation Current

Currently, tactor current is applied in alternating square waves. A current switching scheme that prevents momentary creation of a low resistance path between the power and ground would help reduce tactor switching noise. A simple method to switch the current that avoids any potential power to ground shorting was presented. Other methods that

would provide a more shaped output are under consideration and may actually be used to produce a more sinusoidal current.

3. Programmable Frequency

It is desirable to support many different frequencies because different tactile transmitter designs operate best at specific frequencies. The current TIC design has a frequency selection between 125 Hz and 250 Hz. Incorporating a frequency counter into the tactile design would expand the number of supported frequencies to sixteen.

C. RECOMMENDATIONS FOR NEXT VLSI LAYOUT

1. Elaborate Testing and Measurement Points

The greatest impediment to determining the reason for failure of the current chip is the lack of any test points within the circuit. The next generation chip should include numerous test points throughout all stages of command processing to allow signal tracing. A method currently being considered is the incorporation of eight outputs that provide circuit status information. By coupling these outputs to a four bit input selection, 128 parameters can be monitored to determine chip performance.

2. Timing with Up Counters and Comparators

By using up counters and comparators for system timing, redundant down counters can be eliminated. Counter control would be limited to a single counter while output control would result from comparing the counter value to a stored or calculated value.

D. PROSPECTS FOR FUTURE DEVELOPMENT

1. On-board Current Switching

Due primarily to the complexity of including analog BiCMOS components on a digital CMOS chip, the tactor current switches are housed on a separate chip with the

control signals being provided by the TIC. The final TIC must incorporate these current switches onto the chip to allow embedding the TIC into the tactor casing. This revision is fundamental to the creation of an intelligent tactor.

2. Programmable Addressing

Use of programmable-gates would allow the TIC address to be electronically assigned rather than set using external jumpers. Additionally, multiple address registers could be included to allow each TIC to respond to several different addresses. Multiple addressing would allow implementation of logical groups for more efficient communication.

3. Two-Way Communications

A change to the fundamental system paradigm might incorporate the ability for real-time feedback to the controller. The status data could include all current TIC parameters. Incorporating an onboard vibration sensor could also provide actual indication of tactor operating parameters. Clearly, this change is beyond the early development requirements for a functional tactile interface.

APPENDIX A. TIC MODELING USING VERILOG

A top-down design approach was used to ensure the Tactor Interface Chip performed exactly as required. The Verilog® hardware description language, presented in Reference 6, was used extensively for modeling all TIC components. Components at all abstraction levels were tested using common "test benches" to ensure identical performance between behavioral and structural definitions. Behavioral models were first designed and tested to validate the design descriptions. The elements were then converted to structural designs and tested with the same test bench programs to verify they performed precisely as required.

This appendix documents the Verilog® code used in the TIC design. Each section contains the test bench program, followed by the behavioral design definition and the structural design definition.

A. TACTOR INTERFACE CHIP

```
//*********************************************************************
// File: TIC_test.v
//
// Description:  Test bench for Tactor Interface Chip
//
// Author: Jeff Link
//*********************************************************************

`define PRD 40

module TIC_test;

  reg  [0:7] words [0:17];
  reg  [0:7] send;
  reg  dIn,reset;
  integer ii, jj;        // loop counters
  wire clk,valid;
  wire tPwr1,tPwr2;
```

Figure 42. TIC Test Bench Verilog® source code.

```
  clock #(`PRD/2) clk1 (clk);
//  TIC_b tic (tPwr1, tPwr2, dIn, clk, reset);
  TIC_s tic (tPwr1, tPwr2, dIn, clk, reset);

  initial  begin
    $display("time       \ttPwr1 tPwr2");
    $monitor("time %0d   \t  %b     %b",$time,tPwr1,tPwr2);
    words[ 0]=8'b00011010;  // valid address, execute command
    words[ 1]=8'b10000110;  // put 110 in pw reg & start tactor
    words[ 2]=8'b11000011;  // put 11 in rp reg & start tactor
    words[ 3]=8'b00010010;  // invalid address, ignore command
    words[ 4]=8'b10001010;  // don't put 1010 in pw reg
    words[ 5]=8'b11000100;  // don't put 100 in rp reg
    words[ 6]=8'b00111011;  // invalid address, ignore
    words[ 7]=8'b01111111;  // all call address, execute command
    words[ 8]=8'b00111011;  // invalid address, wait for command
    words[ 9]=8'b10000010;  // put 10 in pw reg & start tactor
    words[10]=8'b11000001;  // put 1 in rp reg & start tactor
    words[11]=8'b00011010;  // valid address, execute command
    words[12]=8'b10000010;  // same pw no need to reload
    words[13]=8'b11000001;  // same rp no need to reload
    words[14]=8'b00011010;  // valid address, execute command
    words[15]=8'b10000000;  // shut off tactor
    words[16]=8'b11000010;  // put 10 in rp reg but won't run
    words[17]=8'b00010010;  // invalid address, ignore command

    reset = 1;
    dIn = 1;                                  // line idle
    #(`PRD/4)
    reset = 0;
    #(2*`PRD)
    for (ii=0; ii<18; ii=ii+1) begin
      send=words[ii];
      #`PRD  dIn = 0;                          // start bit
      for (jj=0; jj<8; jj=jj+1) begin
        #`PRD  dIn=send[jj];                   // data bits
      end
      #`PRD  dIn = ~^send;                     // odd parity
      #`PRD  dIn = 1;                          // stop bit
      $display("time %0d\t sent %b",$time,send);
      #(4*`PRD);
      if (ii==2||ii==5||ii==10||ii==13||ii==16)
        #(1000*`PRD);                          // line idle
    end
    #(100*`PRD)
    $finish;
  end

endmodule
```

Figure 42. TIC Test Bench Verilog® source code (continued)

```
//*********************************************************************
// File: TIC_b.v
//
// Description:  Tactor Interface Chip - behavioral model
//
// Author: Jeff Link
//*********************************************************************

module TIC_b (tPwr1, tPwr2, dIn, clk, reset);

  output tPwr1, tPwr2;
  wire   tPwr1, tPwr2;
  input  dIn, clk, reset;
  wire   [7:0] cmdBus;
  wire   [5:0] pwReg,rpReg;

  ser_rcvr_b   sr (cmdBus, busValid, dIn, clk, reset);
  cmd_decode_b cd (enPwr,pwReg,rpReg,cmdBus,busValid,clk,~reset);
  pwr_cntrl_b  pc (tPwr1, tPwr2, pwReg, rpReg, enPwr, clk);

endmodule
```

Figure 43. TIC Behavioral model Verilog® source code.

```
//*********************************************************************
// File: TIC_s.v
//
// Description:  Tactor Interface Chip - structural model
//
// Author: Jeff Link
//*********************************************************************

module TIC_s (tPwr1, tPwr2, dIn, clk, reset);

  output tPwr1, tPwr2;
  wire   tPwr1, tPwr2;
  input  dIn, clk, reset;
  wire   [7:0] cmdBus;
  wire   [5:0] pwReg,rpReg;

  ser_rcvr_s   sr (cmdBus, busValid, dIn, clk, reset);
  cmd_decode_s cd (enPwr,pwReg,rpReg,cmdBus,busValid,clk,reset);
  pwr_cntrl_s  pc (tPwr1, tPwr2, pwReg, rpReg, enPwr, clk);

endmodule
```

Figure 44. TIC Structural model Verilog® source code.

B. SERIAL DATA RECEIVER

```
//*********************************************************************
// File: ser_rcvr_test.v
//
// Description:  Test bench for Serial Data Receiver
//
// Author: Jeff Link
//*********************************************************************

module ser_rcvr_test;

  reg  [0:7] words [0:9];
  reg  [0:7] send;
  wire [7:0]  v;
  reg  dIn,rst;
  reg  [3:0] ii, jj;       // loop counters
  wire clk,valid;

  clock #(100)   clk1 (clk);
//  ser_rcvr_b rcvr (v,valid,dIn,clk,rst);
  ser_rcvr_s rcvr (v,valid,dIn,clk,rst);

  initial  begin
    words[0]=8'b10101010;
    words[1]=8'b01010101;
    words[2]=8'b11001101;
    words[3]=8'b10110110;
    words[4]=8'b00100100;
    words[5]=8'b11011011;
    words[6]=8'b10111101;
    words[7]=8'b01000010;
    words[8]=8'b00001111;
    words[9]=8'b00111100;
    rst = 1;
    dIn = 1;
    #5
    rst = 0;
    #500
    for (ii=0; ii<10; ii=ii+1) begin
      send=words[ii];
      #200  dIn = 0;                        // start bit
      for (jj=0; jj<8; jj=jj+1) begin
        #200  dIn=send[jj];                 // data bits
      end
      #200  dIn = ~^send;                   // odd parity
      #200  dIn = 1;                        // stop bit
    end
    #400
    $finish;
  end

  always @(valid) begin
    if (valid)
      $display("time %0d   \t  %b is valid",$time,v);
  end

endmodule
```

Figure 45. Serial Data Receiver Test Bench Verilog® source code.

```
//*****************************************************************************
// File: ser_rcvr_b.v
//
// Description:  Serial Data Receiver - behavioral model
//
// Author: Jeff Link
//*****************************************************************************

module ser_rcvr_b (cmdBus, busValid, dIn, clk, reset);
  output [7:0] cmdBus;
  wire   [7:0] cmdBus;
  output busValid;
  input  dIn, clk, reset;
  wire   [11:0] qBus;

  bitshift12_b  bs0 (qBus, dIn, clk, reset, partClear);
  bitlatch8_b   bl0 (cmdBus, qBus[9:2], latch, reset);
  input_valid_b iv0 (latch, busValid, partClear, qBus, clk, reset);

endmodule
```

Figure 46. Serial Data Receiver Behavioral model Verilog® source code.

```
//*****************************************************************************
// File: ser_rcvr_s.v
//
// Description:  Serial Data Receiver - structural model
//
// Author: Jeff Link
//*****************************************************************************

module ser_rcvr_s (cmdBus, busValid, dIn, clk, reset);
  output [7:0] cmdBus;
  wire   [7:0] cmdBus;
  output busValid;
  input  dIn, clk, reset;
  wire   [11:0] qBus;

  bitshift12_s  bs0 (qBus, dIn, clk, reset, partClear);
  bitlatch8_s   bl0 (cmdBus, qBus[9:2], latch, reset);
  input_valid_s iv0 (latch, busValid, partClear, qBus, clk, reset);

endmodule
```

Figure 47. Serial Data Receiver Structural model Verilog® source code.

1. Twelve-Bit Input Shift Register

```
//*********************************************************************
// File: bitshift12_test.v
//
// Description:  Test bench for 12 bit Shift Register
//
// Author: Jeff Link
//*********************************************************************

module bitshift12_test;

  reg  latch,rst;
  reg  [0:7] words [0:1];
  reg  [0:7] send;
  reg  [3:0] ii, jj;       // loop counters
  reg  dIn, reset, partclear;
  wire [11:0] bus;

  clock    clk1 (clk);
  bitshift12_b shift1 (bus, dIn, clk, reset, partclear);
//  bitshift12_s shift1 (bus, dIn, clk, reset, partclear);


  initial  begin
    words[0]=8'b10101010;
    words[1]=8'b10110110;
    $monitor("time %0d  \t%b %b %b %b %b %b %b %b %b %b %b %b is on bus",

$time,bus[11],bus[10],bus[9],bus[8],bus[7],bus[6],bus[5],bus[4],bus[3],bus[2]
,bus[1],bus[0]);
    dIn = 1;
    reset = 1;
    partclear = 0;
    $display("time %0d  \t\t\t\t\t\t     bus reset and idle",$time);
    #2 reset = 0;
    #44
    for (ii=0; ii<2; ii=ii+1) begin
      send=words[ii];
      #9  dIn = 0;                          // start bit
      $display("time %0d  \t\t\t\t\t\t  %b  start bit",$time,dIn);
      for (jj=0; jj<8; jj=jj+1) begin
        #20  dIn=send[jj];                  // data bits
        $display("time %0d  \t\t\t\t\t\t  %b  data bit",$time,dIn);
      end
      #20  dIn = ~^send;                    // odd parity
      $display("time %0d  \t\t\t\t\t\t  %b  parity bit",$time,dIn);
      #20  dIn = 1;                         // stop bit
      $display("time %0d  \t\t\t\t\t\t  %b  stop bit",$time,dIn);
      #9  partclear = 1;
      $display("time %0d  \t\t\t\t\t\t     partial clear",$time);
      #2  partclear = 0;
    end
```

Figure 48. Twelve-Bit Input Shift Register Test Bench Verilog® source code.

```
    #40
    reset=1;
    $display("time %0d  \t\t\t\t\t\t     reset",$time);
    #40
    $finish;
  end

endmodule
```

Figure 48. Twelve-Bit Input Shift Register Test Bench Verilog® source code. (continued)

```
//*********************************************************************
// File: bitshift12_b.v
//
// Description:  12 bit Shift Register - behavioral model
//
// Author: Jeff Link
//*********************************************************************

module bitshift12_b (bus, dIn, clk, reset, partclear);
  output [11:0]  bus;
  reg    [11:0]  bus;
  input  dIn, clk, reset, partclear;
  always  @(posedge clk)  begin
    if (~reset)  begin
      #2
      bus = bus << 1;
      bus[0]  = dIn;
    end
  end

  always  begin
    #1
    if (reset)
      bus = 0;
  end

  always  begin
    #1
    if (partclear & ~reset)
      bus[11:1]  = 0;
  end
endmodule
```

Figure 49. Twelve-Bit Input Shift Register Behavioral model Verilog® source code.

```
//*********************************************************************
// File: bitshift12_s.v
//
// Description:  12 bit Shift Register - structural model
//
// Author: Jeff Link
//*********************************************************************

module bitshift12_s (bus, dIn, clk, reset, partclear);
  output [11:0]  bus;
  wire   [11:0]  bus;
  wire   [11:0]  nbus;
  input  dIn, clk, reset, partclear;
  reg    hi;

  dff_b ds0  (bus[0],nbus[0],dIn,hi,nt1out,clk),
        ds1  (bus[1],nbus[1],bus[0],hi,nr1out,clk),
        ds2  (bus[2],nbus[2],bus[1],hi,nr1out,clk),
        ds3  (bus[3],nbus[3],bus[2],hi,nr1out,clk),
        ds4  (bus[4],nbus[4],bus[3],hi,nr1out,clk),
        ds5  (bus[5],nbus[5],bus[4],hi,nr1out,clk),
        ds6  (bus[6],nbus[6],bus[5],hi,nr1out,clk),
        ds7  (bus[7],nbus[7],bus[6],hi,nr1out,clk),
        ds8  (bus[8],nbus[8],bus[7],hi,nr1out,clk),
        ds9  (bus[9],nbus[9],bus[8],hi,nr1out,clk),
        ds10 (bus[10],nbus[10],bus[9],hi,nr1out,clk),
        ds11 (bus[11],nbus[11],bus[10],hi,nr1out,clk);

  not #1 nt1 (nt1out,reset);

  nor #2 nr1 (nr1out,reset,partclear);

  initial  begin
    hi=1;
  end

endmodule
```

Figure 50. Twelve-Bit Input Shift Register Structural model Verilog® source code.

2. Eight-Bit Data Latch

```
//*********************************************************************
// File: bitlatch8_test.v
//
// Description:  Test bench for 8 bit Data Latch
//
// Author: Jeff Link
//*********************************************************************

module bitlatch8_test;

  reg  latch,rst;
  reg  [8:0] ii;        // loop counter
  wire [7:0] bus;

//  bitlatch8_b latch1 (bus,ii[7:0],latch,rst);
  bitlatch8_s latch1 (bus,ii[7:0],latch,rst);

  initial  begin
    $monitor("time %0d  \t%b %b %b %b %b %b %b %b is latched",
      $time,bus[7],bus[6],bus[5],bus[4],bus[3],bus[2],bus[1],bus[0]);
    rst = 1;
    latch = 0;
    #5
    rst = 0;
    #5;
    for (ii=0; ii<256; ii=ii+17) begin
      $display("time %0d  \t%b %b %b %b %b %b %b %b on bus",
        $time,ii[7],ii[6],ii[5],ii[4],ii[3],ii[2],ii[1],ii[0]);
      #10  latch = 1;
      #10  latch = 0;
      #10  rst = 1;
      #10  rst = 0;
    end
    #40
    $finish;
  end
endmodule
```

Figure 51. Eight-Bit Data Latch Test Bench Verilog® source code.

```
//*****************************************************************************
// File: bitlatch8_b.v
//
// Description:  8 bit Data Latch - behavioral model
//
// Author: Jeff Link
//*****************************************************************************

module bitlatch8_b (bus, inBus, latch, reset);
  output [7:0]  bus;
  reg    [7:0]  bus;
  input  [7:0]  inBus;
  input  latch, reset;

  always  @(posedge latch)  begin
    if (~reset)
      bus = #3 inBus;
  end

  always  begin
    #1
    if (reset)
      bus = 0;
  end
endmodule
```

Figure 52. Eight-Bit Data Latch Behavioral model Verilog® source code.

```
//*****************************************************************************
// File: bitlatch8_s.v
//
// Description:  8 bit Data Latch - structural model
//
// Author: Jeff Link
//*****************************************************************************

module bitlatch8_s (bus, inBus, latch, reset);

  output [7:0]  bus;
  wire   [7:0]  bus;
  wire   [7:0]  nbus;
  input  [7:0]  inBus;
  input  latch, reset;
  reg    hi;

  dff_b db0(bus[0],nbus[0],inBus[0],hi,~reset,latch),
        db1(bus[1],nbus[1],inBus[1],hi,~reset,latch),
        db2(bus[2],nbus[2],inBus[2],hi,~reset,latch),
        db3(bus[3],nbus[3],inBus[3],hi,~reset,latch),
        db4(bus[4],nbus[4],inBus[4],hi,~reset,latch),
        db5(bus[5],nbus[5],inBus[5],hi,~reset,latch),
        db6(bus[6],nbus[6],inBus[6],hi,~reset,latch),
        db7(bus[7],nbus[7],inBus[7],hi,~reset,latch);

  initial  begin
    hi=1;
  end
endmodule
```

Figure 53. Eight-Bit Data Latch Structural model Verilog® source code.

3. Input Stream Validity Check

```
//*********************************************************************
// File: input_valid_test.v
//
// Description:  Test bench for Input Stream Validity Check
//
// Author: Jeff Link
//*********************************************************************

module input_valid_test;

  reg  [0:7] words [0:2];
  reg  [0:7] send;
  reg  [11:0] inBus;
  reg  reset;
  reg  [3:0] ii;        // loop counter

  clock #(100) clk1 (clk);
//  input_valid_b iv (latch, busValid, partClear, inBus, clk, reset);
  input_valid_s iv (latch, busValid, partClear, inBus, clk, reset);

  initial  begin
    $display("\t\t\tlat bV pC inBus");
    $monitor("time %0d  \t %b  %b  %b   %b",
      $time,latch,busValid,partClear,inBus);
    words[0]=8'b10101010;
    words[1]=8'b10110110;
    words[2]=8'b01000010;
    reset = 0;
    #25
    for (ii=0; ii<3; ii=ii+1) begin
      inBus[11]=1;
      inBus[10]=0;
      inBus[9:2]=words[ii];
      inBus[1]=~^words[ii];
      inBus[0]=1;
      #200;
    end
    #400
    $finish;
  end

  always @(busValid) begin
    if (busValid)
      $display("time %0d\t    %b is valid",$time,inBus);
  end

  always @(posedge partClear) begin
    #1
    inBus[11:1]=0;
    #40
  inBus[10]=1;
  end
endmodule
```

Figure 54. Input Stream Validity Check Test Bench Verilog® source code.

```
//*****************************************************************************
// File: input_valid_b.v
//
// Description:  Input Stream Validity Check - behavioral model
//
// Author: Jeff Link
//*****************************************************************************

module input_valid_b (latch, busValid, partClear, inBus, clk, reset);
  output latch, busValid, partClear;
  reg    latch, busValid, partClear;
  input  [11:0] inBus;
  input  clk, reset;
  reg    format,clearValid;

  initial begin
    latch = 0;
    busValid = 0;
    partClear = 0;
    clearValid=0;
  end

  always begin
    #1
    format = #12 (inBus[11]&~inBus[10]&inBus[0]&(^inBus[9:1]));
    clearValid = (~inBus[11]&inBus[10]&busValid);
  end

  always begin
    #1;
    latch = #2 (~(~format | clk));
  end

  always begin
    #1
    partClear = #2 (format & busValid);
  end

  always  @(posedge latch)  begin
    if (~(reset | clearValid))  begin
      busValid = #3 format;
    end
  end

  always  begin
    #1
    if (reset | clearValid)  begin
      busValid = #1 0;
    end
  end
endmodule
```

Figure 55. Input Stream Validity Check Behavioral model Verilog® source code.

```
//*********************************************************************
// File: input_valid_s.v
//
// Description:  Input Stream Validity Check - structural model
//
// Author: Jeff Link
//*********************************************************************

module input_valid_s (latch, busValid, partClear, inBus, clk, reset);
  output latch, busValid, partClear;
  wire   latch, busValid, partClear;
  input  [11:0] inBus;
  input  clk, reset;
  reg    hi;

//  nor #2  nrbus(nrbusout,nBusLatch,valid,clk);

  xor #2  xrp0(xrp0out,inBus[1],inBus[2]),   // parity check
          xrp1(xrp1out,inBus[3],inBus[4]),
          xrp2(xrp2out,inBus[5],inBus[6]),
          xrp3(xrp3out,inBus[7],inBus[8]),
          xrp4(xrp4out,xrp0out,xrp1out),
          xrp5(xrp5out,xrp3out,inBus[9]),
          xrp6(xrp6out,xrp2out,xrp5out),
          xrp7(parity,xrp4out,xrp6out);

  not #1  nt0(ninBus10,inBus[10]),
          nt1(nformat,format);

  nand #2 naf0(naf0out,parity,ninBus10),   // format check
          naf1(naf1out,inBus[11],inBus[0]);
  nor #2  nrf0(format,naf0out,naf1out),
          nrl1(latch,nformat,clk),           // latch
          nrr2(nclrDff,reset,clrValid);      // clear bus valid

  and #2  and0(partClear,format,busValid);   // partial clear

  nor #2  nr0(clrValid,inBus[11],ninBus10,nbusValid); // clear bus valid

  dff_b   db0(busValid,nbusValid,format,hi,nclrDff,latch);// busValid

  initial
    hi=1;

endmodule
```

Figure 56. Input Stream Validity Check Structural model Verilog® source code.

C. COMMAND DECODER AND CONTROLLER

```
//*********************************************************************
// File: cmd_decode_test.v
//
// Description:  Test bench for Command Decoder and Controller
//
// Author: Jeff Link
//*********************************************************************

module cmd_decode_test;

  reg  [7:0] cmdBus;
  reg  busValid,reset;

  wire   [5:0] pwReg,rpReg;

  clock   clk1 (clk);
//  cmd_decode_b dc0 (enPwr,pwReg,rpReg,cmdBus,busValid,clk,reset);
  cmd_decode_s dc1 (enPwr,pwReg,rpReg,cmdBus,busValid,clk,reset);

  initial  begin
    $display("\t\t\tenPwr  pwReg   rpReg");
    $monitor("time %0d   \t  %b    %b  %b",
      $time,enPwr,pwReg,rpReg);
    cmdBus=0;
    busValid=0;
    reset=1;
    #7  reset=0;
    #9  cmdBus=8'b00011010;  // valid address, execute command
        busValid=1;  // 16
    #80 cmdBus=8'b10000110;  // put 110 in pw reg & start tactor
        busValid=1;  // 96
    #80 cmdBus=8'b11000011;  // put 11 in rp reg & start tactor
        busValid=1;  //176
    #80 cmdBus=8'b00010010;  // invalid address, ignore command
        busValid=1;  //256
    #80 cmdBus=8'b10001010;  // don't put 1010 in pw reg
        busValid=1;  //336
    #80 cmdBus=8'b11000100;  // don't put 100 in rp reg
        busValid=1;  //416
    #80 cmdBus=8'b01111111;  // all call address, execute command
        busValid=1;  //496
    #80 cmdBus=8'b10001010;  // put 1010 in pw reg & start tactor
        busValid=1;  //576
    #80 cmdBus=8'b11001011;  // put 1011 in rp reg & start tactor
        busValid=1;  //656
    #80 cmdBus=8'b00011010;  // valid address, execute command
        busValid=1;  //736
    #80 cmdBus=8'b10000000;  // put 0000 in pw reg & start tactor
        busValid=1;  //816
    #80 cmdBus=8'b11000010;  // put 0010 in rp reg & start tactor
        busValid=1;  //896
    #80 cmdBus=8'b00010010;  // invalid address, ignore command
        busValid=1;  //976
    #100
    $finish
  end
```

Figure 57. Command Decoder and Controller Test Bench Verilog® source code.

```
  always @ (cmdBus) begin
    #75 busValid=0;
  end

endmodule
```

Figure 57. Command Decoder and Controller Test Bench Verilog® source code. (continued)

```
//*****************************************************************************
// File: cmd_decode_b.v
//
// Description:  Command Decoder and Controller - behavioral model
//
// Author: Jeff Link
//*****************************************************************************

module cmd_decode_b (enPwr,pwReg,rpReg,cmdBus,busValid,clk,nReset);

  output enPwr,pwReg,rpReg;
  wire   enPwr;
  wire   [5:0] pwReg,rpReg;
  input  [7:0] cmdBus;
  input  busValid,clk,nReset;
  wire   [6:0] tactAddr;

  addr_ref_b  ar (tactAddr);
  addr_comp_b ac (validAddr,cmdBus,tactAddr);
  cmd_logic_b cl (enPwr,pwLatch,rpLatch,
                  cmdBus,busValid,validAddr,pwDiff,rpDiff,clk,nReset);
  pw_reg_b    pr (pwReg,pwDiff,cmdBus[5:0],pwLatch,nReset);
  rp_reg_b    rr (rpReg,rpDiff,cmdBus[5:0],rpLatch,nReset);

endmodule
```

Figure 58. Command Decoder and Controller Behavioral model Verilog® source code.

```
//*****************************************************************************
// File: cmd_decode_s.v
//
// Description:  Command Decoder and Controller - structural model
//
// Author: Jeff Link
//*****************************************************************************

module cmd_decode_s (enPwr,pwReg,rpReg,cmdBus,busValid,clk,reset);

  output enPwr,pwReg,rpReg;
  wire   enPwr;
  wire   [5:0] pwReg,rpReg;
  input  [7:0] cmdBus;
  input  busValid,clk,reset;
  wire   [6:0] tactAddr;

addr_ref_b  ar (tactAddr);
addr_comp_s ac (validAddr,cmdBus,tactAddr);
cmd_logic_s cl (enPwr,pwLatch,rpLatch,
                cmdBus,busValid,validAddr,pwDiff,rpDiff,clk,~reset);
pw_reg_s    pr (pwReg,pwDiff,cmdBus[5:0],pwLatch,reset);
rp_reg_s    rr (rpReg,rpDiff,cmdBus[5:0],rpLatch,reset);

endmodule
```

Figure 59. Command Decoder and Controller Structural model Verilog® source code.

1. Command Sequence Controller

```
//*********************************************************************
// File: cmd_logic_test.v
//
// Description:  Test bench for Command Sequence Controller
//
// Author: Jeff Link
//*********************************************************************

module cmd_logic_test;

  reg  [7:0] cmdBus;
  reg  busValid,nValidAddr,pwDiff,rpDiff,nReset;

  clock   clk1 (clk);
//  cmd_logic_b log  (enPwr,pwLatch,rpLatch,
//                    cmdBus,busValid,nValidAddr,pwDiff,rpDiff,clk,nReset);
  cmd_logic_s log  (enPwr,pwLatch,rpLatch,
                    cmdBus,busValid,nValidAddr,pwDiff,rpDiff,clk,nReset);

  initial  begin
    $display("\t\t\tenPwr pwLatch rpLatch");
    $monitor("time %0d   \t  %b     %b        %b",
      $time,enPwr,pwLatch,rpLatch);
    cmdBus=0;
    busValid=0;
    nValidAddr=1;
    pwDiff=0;
    rpDiff=0;
    nReset=0;
    #7  nReset=1;
    #9  cmdBus=8'b00011010;
        busValid=1;  //16
    #5  nValidAddr=0;     //21
    #21 busValid=0;  //42
    #26 nValidAddr=1;
        cmdBus=8'b10000110;
        busValid=1;  //68
    #4  pwDiff=1;    //72
    #26 busValid=0;  //98
    #12 cmdBus=8'b11000010;
        busValid=1;  //110
    #4  rpDiff=1;    //114
    #19 busValid=0;  //133
    #41 cmdBus=8'b00000000;
        busValid=1;  //174
    #4  pwDiff=1;    //178
    #12 busValid=0;  //190
    #4  cmdBus=8'b00000000;
        busValid=1;  //194
    #16 busValid=0;  //210
    #100
    $finish;
  end
```

Figure 60. Command Sequence Controller Test Bench Verilog® source code.

```
  always @ (posedge pwLatch) begin
    #3 pwDiff=0;
  end

  always @ (posedge rpLatch) begin
    #3 rpDiff=0;
  end

endmodule
```

Figure 60. Command Sequence Controller Test Bench Verilog® source code. (continued)

```
//*****************************************************************************
// File: cmd_logic_b.v
//
// Description:  Command Sequence Controller - behavioral model
//
// Author: Jeff Link
//*****************************************************************************

module cmd_logic_b (enPwr,pwLatch,rpLatch,
                    cmdBus,busValid,vAddr,pwDiff,rpDiff,clk,nReset);
  output enPwr,pwLatch,rpLatch;
  reg    enPwr,pwLatch,rpLatch;
  reg    [1:0] state;
  input  [7:0] cmdBus;
  input  busValid,vAddr,pwDiff,rpDiff,clk,nReset;

  initial begin
    state=0;
    enPwr=0;
    pwLatch=0;
    rpLatch=0;
  end

  always @ (posedge clk) begin
    if (nReset) begin
      case(state)
        2'b00,2'b10:
          if (busValid&~cmdBus[7]&vAddr)
            state=1;
        2'b01:
          if (busValid&cmdBus[7])
            state=3;
        2'b11:
          if (busValid&~cmdBus[7])
            state=0;
      endcase
    end
  end
```

Figure 61. Command Sequence Controller Behavioral model Verilog® source code.

```
  always begin
    if (nReset&&state==3) begin
      if (cmdBus==8'b10000000&&busValid)
        enPwr=0;
      else if (cmdBus[7]&&~cmdBus[6]&&busValid)
        enPwr=~pwDiff;
      else if (cmdBus[7]&&cmdBus[6]&&busValid)
        enPwr=~rpDiff;
      pwLatch=(cmdBus[7]&&~cmdBus[6]&&busValid&&pwDiff);
      rpLatch=(cmdBus[7]&&cmdBus[6]&&busValid&&rpDiff);
    end
    #1;
  end

  always @ (nReset) begin
    if (~nReset) begin
      state=0;
      enPwr=0;
      pwLatch=0;
      rpLatch=0;
    end
  end

endmodule
```

Figure 61. Command Sequence Controller Behavioral model Verilog® source code. (continued)

```
//*****************************************************************************
// File: cmd_logic_s.v
//
// Description:  Command Sequence Controller - structural model
//
// Author: Jeff Link
//*****************************************************************************

module cmd_logic_s (enPwr,pwLatch,rpLatch,
                    cmdBus,busValid,nValidAddr,pwDiff,rpDiff,clk,nReset);
  output enPwr,pwLatch,rpLatch;
  wire   [1:0] q,nq;
  input  [7:0] cmdBus;
  input  busValid,nValidAddr,pwDiff,rpDiff,clk,nReset;
  reg    hi;

  dff_b   ds0(q[0],nq[0],nd2out,hi,nReset,clk),    // state registers
          ds1(q[1],nq[1],nt0out,hi,nReset,clk);

  not #1  nt0(nt0out,nd0out),
          nt4(nt4out,cmdBus[6]),
          nt5(pwLatch,nd4out),
          nt6(rpLatch,nd5out);

  nand #2 nd0(nd0out,cmdBus[7],q[0]),
          nd1(nd1out,nq[1],q[0]),
          nd2(nd2out,nd0out,nd1out,or0out),
          nd3(nd3out,nr0out,nr1out,an0out),
          nd4(nd4out,an1out,pwDiff,nt4out),
          nd5(nd5out,an1out,rpDiff,cmdBus[6]),
          nd6(nd6out,nReset,nd3out),
          nd7(nd7out,an1out,pwDiff,rpDiff),
          nd8(nd8out,nd4out,nd5out,nd7out);

  or  #2  or0(or0out,cmdBus[7],q[0],nValidAddr),
          or1(or1out,nd6out,nd8out);

  nor  #2 nr0(nr0out,cmdBus[0],cmdBus[1],cmdBus[2]),
          nr1(nr1out,cmdBus[3],cmdBus[4],cmdBus[5]),
          nr6(enPwr,nr7out,or1out),
          nr7(nr7out,enPwr,an1out);

  and #2  an0(an0out,cmdBus[7],busValid,nt4out),
          an1(an1out,q[1],q[0],busValid,cmdBus[7]);

  initial
    hi=1;
endmodule
```

Figure 62. Command Sequence Controller Structural model Verilog® source code.

2. Address Comparator

```
//*********************************************************************
// File: addr_comp_test.v
//
// Description:  Test bench for Address Comparator
//
// Author: Jeff Link
//*********************************************************************

module addr_comp_test;

  reg  [7:0] inBus;
  wire [6:0] tactAddr;

  addr_ref_b  addref (tactAddr);
//  addr_comp_b addcmp (nValidAddr,inBus,tactAddr);
  addr_comp_s addcmp (nValidAddr,inBus,tactAddr);
//  addr_comp_alt addcmp (nValidAddr,inBus,tactAddr);

  initial  begin
    for (inBus=0; inBus<255; inBus=inBus+1) begin
      #20
      if (~nValidAddr)
        $display("time %0d\t    %b is valid address",$time,inBus);
    end
    #20
    if (~nValidAddr)
      $display("time %0d\t    %b is valid address",$time,inBus);
    $finish;
  end
endmodule
```

Figure 63. Address Comparator Test Bench Verilog® source code.

```
//*********************************************************************
// File: addr_comp_b.v
//
// Description:  Address Comparator - behavioral model
//
// Author: Jeff Link
//*********************************************************************


module addr_comp_b (validAddr,inBus,tactAddr);
  output validAddr;
  reg    validAddr;
  input  [7:0] inBus;
  input  [6:0] tactAddr;

  always begin
    #4
    validAddr = (~inBus[7]&&(inBus[6:0]==tactAddr||inBus[6:0]==7'b1111111));
  end
endmodule
```

Figure 64. Address Comparator Behavioral model Verilog® source code.

```
//*********************************************************************
// File: addr_comp_s.v
//
// Description:  Address Comparator - structural model
//
// Author: Jeff Link
//*********************************************************************


module addr_comp_s (nValidAddr,inBus,tactAddr);
  output nValidAddr;
  input  [7:0] inBus;
  input  [6:0] tactAddr;


  xnor #2 xeq0(xeq0out,inBus[0],tactAddr[0]),   // match reference
          xeq1(xeq1out,inBus[1],tactAddr[1]),
          xeq2(xeq2out,inBus[2],tactAddr[2]),
          xeq3(xeq3out,inBus[3],tactAddr[3]),
          xeq4(xeq4out,inBus[4],tactAddr[4]),
          xeq5(xeq5out,inBus[5],tactAddr[5]),
          xeq6(xeq6out,inBus[6],tactAddr[6]);
  nand #2 nae0(nae0out,xeq0out,xeq1out,xeq2out),
          nae2(nae2out,xeq3out,xeq4out,xeq5out),
          nae3(nae3out,xeq6out,ninBus7);
  nor #2  nre0(thisAddr,nae0out,nae2out,nae3out);

  not #1  nte0(ninBus7,inBus[7]);            // common elements
  nor #2  nrn0(nValidAddr,thisAddr,allCall);

  nand #2 naa0(naa0out,inBus[0],inBus[1],inBus[2]),   // all call check
          naa2(naa2out,inBus[3],inBus[4],inBus[5]),
          naa3(naa3out,inBus[6],ninBus7);
  nor #2  nra0(allCall,naa0out,naa2out,naa3out);

endmodule
```

Figure 65. Address Comparator Structural model Verilog® source code.

```
//*****************************************************************************
// File: addr_comp_alt.v
//
// Description:  Address Comparator - alternate model
//
// Author: Jeff Link
//*****************************************************************************


module addr_comp_alt (validAddr,inBus,tactAddr);
  output validAddr;
  input  [7:0] inBus;
  input  [6:0] tactAddr;

  xnor #2 xeq0(xeq0out,inBus[0],tactAddr[0]),   // match reference
          xeq1(xeq1out,inBus[1],tactAddr[1]),
          xeq2(xeq2out,inBus[2],tactAddr[2]),
          xeq3(xeq3out,inBus[3],tactAddr[3]),
          xeq4(xeq4out,inBus[4],tactAddr[4]),
          xeq5(xeq5out,inBus[5],tactAddr[5]),
          xeq6(xeq6out,inBus[6],tactAddr[6]);
  and #2  ane0(ane0out,xeq0out,xeq1out),
          ane1(ane1out,xeq2out,xeq3out),
          ane2(ane2out,xeq4out,xeq5out),
          ane3(ane3out,xeq6out,ntc0out),
          ane4(ane4out,ane0out,ane1out),
          ane5(ane5out,ane2out,ane3out),
          ane6(thisAddr,ane4out,ane5out);

  not #1  ntc0(ntc0out,inBus[7]);            // common elements
  or #2   orc0(validAddr,thisAddr,allCall);

  and #2  ana0(ana0out,inBus[0],inBus[1]),   // all call check
          ana1(ana1out,inBus[2],inBus[3]),
          ana2(ana2out,inBus[4],inBus[5]),
          ana3(ana3out,inBus[6],ntc0out),
          ana4(ana4out,ana0out,ana1out),
          ana5(ana5out,ana2out,ana3out),
          ana6(allCall,ana4out,ana5out);

endmodule
```

Figure 66. Address Comparator Alternate Structural model Verilog® source code.

3. Address Reference

```
//*****************************************************************************
// File: addr_ref_test.v
//
// Description:  Test bench for Address Reference
//
// Author: Jeff Link
//*****************************************************************************

module addr_ref_test;
  wire   [6:0] tactAddr;
  addr_ref_b addr1 (tactAddr);

  initial  begin
    #1
    $display("time %0d\t  %b is reference address",$time,tactAddr);
    $finish;
  end
endmodule
```

Figure 67. Address Reference Test Bench Verilog® source code.

```
//*****************************************************************************
// File: addr_ref_b.v
//
// Description:  Address Reference - behavioral model
//
// Author: Jeff Link
//*****************************************************************************

`define ADDRESS 7'b0011010

module addr_ref_b (tactAddr);
  output [6:0] tactAddr;
  reg    [6:0] tactAddr;

  initial begin
    tactAddr = `ADDRESS;
    #1;
  end
endmodule
```

Figure 68. Address Reference Behavioral model Verilog® source code.

4. Pulse Width Register

```
//*********************************************************************
// File: pw_reg_test.v
//
// Description:  Test bench for Pulse Width Register
//
// Author: Jeff Link
//*********************************************************************

module pw_reg_test;

  reg  [5:0] inBus;
  reg  latch,reset;
  wire [5:0] pwReg;

//  pw_reg_b reg1 (pwReg,pwDiff,inBus,latch,reset);
  pw_reg_s reg1 (pwReg,pwDiff,inBus,latch,reset);

  initial  begin
    $display("\t\t\t  inBus   pwReg  pwDiff");
    reset=1;
    #1
    reset=0;
    #1
    latch=1;
    #3
    for (inBus=2; inBus<63; inBus=inBus+5) begin
      #4
      latch=~latch;
      #20
      latch=~latch;
      #16;
    end
    #100
    $finish;
  end

  always @ (pwDiff) begin
    $display("time %0d  \t  %b  %b  %b",$time,inBus,pwReg,pwDiff);
  end

endmodule
```

Figure 69. Pulse Width Register Test Bench Verilog® source code.

```
//*****************************************************************************
// File: pw_reg_b.v
//
// Description:  Pulse Width Register - behavioral model
//
// Author: Jeff Link
//*****************************************************************************

module pw_reg_b (pwReg,pwDiff,inBus,latch,nReset);

  output [5:0] pwReg;
  output pwDiff;
  reg    [5:0] pwReg;
  reg    pwDiff;
  input  [5:0] inBus;
  input  latch,nReset;

  always @ (posedge latch) begin
    if (nReset)
      #3 pwReg=inBus;
  end

  always @ (inBus) begin
    if (nReset)
      #4 pwDiff=(inBus!=pwReg);
  end

  always @ (pwReg) begin
    if (nReset)
      #4 pwDiff=(inBus!=pwReg);
  end

  always @ (negedge nReset) begin
    #1 pwReg=0;
  end
endmodule
```

Figure 70. Pulse Width Register Behavioral model Verilog® source code.

```
//*****************************************************************************
// File: pw_reg_s.v
//
// Description:  Pulse Width Register - structural model
//
// Author: Jeff Link
//*****************************************************************************

module pw_reg_s (pwReg,pwDiff,inBus,latch,reset);

  output [5:0] pwReg;
  output pwDiff;
  wire   [5:0] pwReg,npwReg;
  input  [5:0] inBus;
  input  latch,reset;

  reg    hi;

  dff_b db0(pwReg[0],npwReg[0],inBus[0],hi,~reset,latch),
        db1(pwReg[1],npwReg[1],inBus[1],hi,~reset,latch),
        db2(pwReg[2],npwReg[2],inBus[2],hi,~reset,latch),
        db3(pwReg[3],npwReg[3],inBus[3],hi,~reset,latch),
        db4(pwReg[4],npwReg[4],inBus[4],hi,~reset,latch),
        db5(pwReg[5],npwReg[5],inBus[5],hi,~reset,latch);

  xor #2  xeq0(xeq0out,inBus[0],pwReg[0]),
          xeq1(xeq1out,inBus[1],pwReg[1]),
          xeq2(xeq2out,inBus[2],pwReg[2]),
          xeq3(xeq3out,inBus[3],pwReg[3]),
          xeq4(xeq4out,inBus[4],pwReg[4]),
          xeq5(xeq5out,inBus[5],pwReg[5]);
  nor #2  nr0(nr0out,xeq0out,xeq1out,xeq2out),
          nr1(nr1out,xeq3out,xeq4out,xeq5out);

  nand #2 nd0(pwDiff,nr0out,nr1out);

  initial  begin
    hi=1;
  end

endmodule
```

Figure 71. Pulse Width Register Structural model Verilog® source code.

5. Repetition Period Register

```
//*********************************************************************
// File: rp_reg_test.v
//
// Description:  Test bench for Repetition Period Register
//
// Author: Jeff Link
//*********************************************************************

module rp_reg_test;

  reg  [5:0] inBus;
  reg  latch,reset;
  wire [5:0] rpReg;

//  rp_reg_b reg1 (rpReg,rpDiff,inBus,latch,reset);
  rp_reg_s reg1 (rpReg,rpDiff,inBus,latch,reset);

  initial  begin
    $display("\t\t\t  inBus   rpReg  rpDiff");
    reset=1;
    #1
    reset=0;
    #1
    latch=1;
    #3
    for (inBus=2; inBus<63; inBus=inBus+5) begin
      #4
      latch=~latch;
      #20
      latch=~latch;
      #16;
    end
    #100
    $finish;
  end

  always @ (rpDiff) begin
    $display("time %0d  \t  %b  %b  %b",$time,inBus,rpReg,rpDiff);
  end

endmodule
```

Figure 72. Repetition Period Register Test Bench Verilog® source code.

```
//*********************************************************************
// File: rp_reg_b.v
//
// Description:  Repetition Period Register - behavioral model
//
// Author: Jeff Link
//*********************************************************************

module rp_reg_b (rpReg,rpDiff,inBus,latch,nReset);

  output [5:0] rpReg;
  output rpDiff;
  reg    [5:0] rpReg;
  reg    rpDiff;
  input  [5:0] inBus;
  input  latch,nReset;

  always @ (posedge latch) begin
    if (nReset)
      #3 rpReg=inBus;
  end

  always @ (inBus) begin
    if (nReset)
      #4 rpDiff=(inBus!=rpReg);
  end

  always @ (rpReg) begin
    if (nReset)
      #4 rpDiff=(inBus!=rpReg);
  end

  always @ (negedge nReset) begin
    #1 rpReg=0;
  end
endmodule
```

Figure 73. Repetition Period Register Behavioral model Verilog® source code.

```
//*********************************************************************
// File: rp_reg_s.v
//
// Description:  Repetition Period Register - structural model
//
// Author: Jeff Link
//*********************************************************************

module rp_reg_s (rpReg,rpDiff,inBus,latch,reset);

  output [5:0] rpReg;
  output rpDiff;
  wire   [5:0] rpReg,nrpReg;
  input  [5:0] inBus;
  input  latch,reset;

  reg    hi;

  dff_b db0(rpReg[0],nrpReg[0],inBus[0],hi,~reset,latch),
        db1(rpReg[1],nrpReg[1],inBus[1],hi,~reset,latch),
        db2(rpReg[2],nrpReg[2],inBus[2],hi,~reset,latch),
        db3(rpReg[3],nrpReg[3],inBus[3],hi,~reset,latch),
        db4(rpReg[4],nrpReg[4],inBus[4],hi,~reset,latch),
        db5(rpReg[5],nrpReg[5],inBus[5],hi,~reset,latch);

  xor #2  xeq0(xeq0out,inBus[0],rpReg[0]),
          xeq1(xeq1out,inBus[1],rpReg[1]),
          xeq2(xeq2out,inBus[2],rpReg[2]),
          xeq3(xeq3out,inBus[3],rpReg[3]),
          xeq4(xeq4out,inBus[4],rpReg[4]),
          xeq5(xeq5out,inBus[5],rpReg[5]);
  nor #2  nr0(nr0out,xeq0out,xeq1out,xeq2out),
          nr1(nr1out,xeq3out,xeq4out,xeq5out);

  nand #2 nd0(rpDiff,nr0out,nr1out);

  initial  begin
    hi=1;
  end

endmodule
```

Figure 74. Repetition Period Register Structural model Verilog® source code.

D. TACTOR POWER CONTROLLER

```
//*********************************************************************
// File: pwr_cntrl_test.v
//
// Description:  Test bench for Tactor Power Controller
//
// Author: Jeff Link
//*********************************************************************

module pwr_cntrl_test;

  reg  [5:0] pwReg,rpReg;
  reg  tEnable;

  clock #(25) clk1 (clk);
//  pwr_cntrl_b pc0 (tPwr1, tPwr2, pwReg, rpReg, tEnable, clk);
  pwr_cntrl_s pc0 (tPwr1, tPwr2, pwReg, rpReg, tEnable, clk);

  initial  begin
    tEnable=0;
    pwReg=3;
    rpReg=2;
    $display("\t  pwReg = %b   rpReg =  %b\n",pwReg,rpReg);
    $display("\t\t\ttPwr1 tPwr2");
    $monitor("time %0d  \t  %b   %b",$time,tPwr1,tPwr2);
    #2000
    tEnable=1;
    #2000000
    $finish;
  end
endmodule
```

Figure 75. Tactor Power Controller Test Bench Verilog® source code.

```
//*********************************************************************
// File: pwr_cntrl_b.v
//
// Description:  Tactor Power Controller - behavioral model
//
// Author: Jeff Link
//*********************************************************************

module pwr_cntrl_b (tPwr1, tPwr2, pwReg, rpReg, tEnable, clk);

  output tPwr1,tPwr2;
  wire   tPwr1,tPwr2;
  input  [5:0] pwReg, rpReg;
  input  tEnable, clk;

  clk_div_b   clkdiv (fr250, fr62, clk, ~tEnable);
  pw_dncntr_b pwCntr (pwZero,pwReg,cntLd,cntClr,fr62);
  rp_dncntr_b rpCntr (rpGT1 ,rpReg,cntLd,cntClr,fr62);
  pwr_logic_b pLogic (enPwr,cntLd,cntClr,tEnable,pwZero,rpGT1,fr62);
  pwr_osc_b   pOscil (tPwr1, tPwr2, enPwr, fr250);

endmodule
```

Figure 76. Tactor Power Controller Behavioral model Verilog® source code.

```
//*********************************************************************
// File: pwr_cntrl_s.v
//
// Description:  Tactor Power Controller - structural model
//
// Author: Jeff Link
//*********************************************************************

module pwr_cntrl_s (tPwr1, tPwr2, pwReg, rpReg, tEnable, clk);

  output tPwr1,tPwr2;
  input  [5:0] pwReg, rpReg;
  input  tEnable, clk;

  clk_div_s   clkdiv (fr250, fr62, clk,reset);
  pw_dncntr_s pwCntr (npwZero,pwReg,cntLd,cntClr,fr62);
  rp_dncntr_s rpCntr (nrpGT1 ,rpReg,cntLd,cntClr,fr62);
  pwr_logic_s pLogic (enPwr,cntLd,cntClr,tEnable,npwZero,nrpGT1,fr62);
  pwr_osc_s   pOscil (tPwr1, tPwr2, enPwr, fr250);

  not #1 nt0 (reset,tEnable);

endmodule
```

Figure 77. Tactor Power Controller Structural model Verilog® source code.

1. Power Control Logic

```
//*********************************************************************
// File: pwr_logic_test.v
//
// Description:  Test bench for Power Control Logic
//
// Author: Jeff Link
//*********************************************************************

module pwr_logic_test;

  reg  enable,npwZero,nrpGT1;

  clock  clk1 (clk);
//  pwr_logic_b pl0 (enPwr,cntLd,cntClr,enable,npwZero,nrpGT1,clk);
  pwr_logic_s pl0 (enPwr,cntLd,cntClr,enable,npwZero,nrpGT1,clk);

  initial  begin
    enable=0;
    npwZero=0;
    nrpGT1=1;
    $display("\t\t\tenPwr cntLd cntClr");
    $monitor("time %0d  \t  %b     %b     %b",$time,enPwr,cntLd,cntClr);
    #25
    enable=1;  // 25 - 110
    #40
    npwZero=1;  // 65 - 100
    #20
    nrpGT1=0;   // 85 - 101
    #20
    npwZero=0;  //105 - 111
    #20
    nrpGT1=1;   //125 - 110
    #20
    npwZero=1;
    nrpGT1=0;   //145 - 101
    #20
    nrpGT1=1;   //165 - 100
    #20
    npwZero=0;  //185 - 111
    nrpGT1=0;
    #40;
    $finish;
  end
endmodule
```

Figure 78. Power Control Logic Test Bench Verilog® source code.

```
//*****************************************************************************
// File: pwr_logic_b.v
//
// Description:  Power Control Logic - behavioral model
//
// Author: Jeff Link
//*****************************************************************************

module pwr_logic_b (enPwr,cntLd,cntClr,enable,pwZero,rpGT1,clk);

  output enPwr,cntLd,cntClr;
  reg    enPwr,cntLd,cntClr;
  input  enable,pwZero,rpGT1,clk;

  always begin
    cntClr=~enable;
    enPwr=~pwZero;
    cntLd=~rpGT1;
    #1;
  end

endmodule
```

Figure 79. Power Control Logic Behavioral model Verilog® source code.

```
//*****************************************************************************
// File: pwr_logic_s.v
//
// Description:  Power Control Logic - structural model
//
// Author: Jeff Link
//*****************************************************************************

module pwr_logic_s (enPwr,cntLd,cntClr,enable,npwZero,nrpGT1,clk);

  output enPwr,cntLd,cntClr;
  reg    enPwr,cntLd;
  input  enable,npwZero,nrpGT1,clk;

  not #1 nt0(cntClr,enable);

  always begin
    enPwr=npwZero;
    cntLd=nrpGT1;
    #1;
  end

endmodule
```

Figure 80. Power Control Logic Structural model Verilog® source code.

2. Power Oscillator

```
//*********************************************************************
// File: pwr_osc_test.v
//
// Description:  Test bench for Power Oscillator
//
// Author: Jeff Link
//*********************************************************************

module clk_div_test;

  reg     enable;
  integer ii;       // loop counter

  clock #(50) clk0 (clk);
//  pwr_osc_b   po0  (pwr1, pwr2, enable, clk);
  pwr_osc_s   po0  (pwr1, pwr2, enable, clk);

  initial  begin
    $display("          \tpwr1 pwr2 enable");
    $monitor("time %0d  \t  %b    %b    %b",$time,pwr1,pwr2,enable);
    enable=1;
    ii = 0;
    while (ii<12) begin
      #5;
      if (ii%3==2)
        enable=0;
      else
        enable=1;
    end
    $finish;
  end

  always @ (posedge clk)
    ii=ii+1;

endmodule
```

Figure 81. Power Oscillator Test Bench Verilog® source code.

```
//*****************************************************************************
// File: pwr_osc_b.v
//
// Description:  Power Oscillator - behavioral model
//
// Author: Jeff Link
//*****************************************************************************

module pwr_osc_b (pwr1, pwr2, enable, osc);
  output pwr1, pwr2;
  reg    pwr1, pwr2;
  input  enable, osc;

  always  begin
    #2
    if (~enable) begin
      pwr1=0;
      pwr2=0;
    end
    else begin
      pwr1=osc;
      pwr2=~osc;
    end
  end

endmodule
```

Figure 82. Power Oscillator Behavioral model Verilog® source code.

```
//*****************************************************************************
// File: pwr_osc_s.v
//
// Description:  Power Oscillator - structural model
//
// Author: Jeff Link
//*****************************************************************************

module pwr_osc_s (pwr1, pwr2, enable, osc);
  output pwr1, pwr2;
  wire   pwr1, pwr2;
  input  enable, osc;

  nand #2 (npwr1,enable,osc),
          (npwr2,enable,nosc);

  not     (nosc,osc),
          (pwr1,npwr1),
          (pwr2,npwr2);

endmodule
```

Figure 83. Power Oscillator Structural model Verilog® source code.

3. Pulse Width Down Counter

```
//*********************************************************************
// File: pw_dncntr_test.v
//
// Description:  Test bench for Pulse Width Down Counter
//
// Author: Jeff Link
//*********************************************************************

module pw_dncntr_test;

  reg  [5:0] value;
  reg  load,clear;

  clock    clk1 (clk);
//  pw_dncntr_b down (npwZero,value,load,clear,clk);
  pw_dncntr_s down (npwZero,value,load,clear,clk);

  initial  begin
    value=9;
    load=0;
    clear=1;
    $display("\t\t\t   nZro clear load");
    $monitor("time %0d  \t    %b    %b     %b",$time,npwZero,clear,load);
    #25
    load=1;
    #40
    clear=0;  // 65
    #100
    load=0;  // 165
    #100
    load=1;  // 265
    #80
    load=0;  // 345
    #400     // 745
    $finish;
  end
endmodule
```

Figure 84. Pulse Width Down Counter Test Bench Verilog® source code.

```
//*****************************************************************************
// File: pw_dncntr_b.v
//
// Description:  Pulse Width Down Counter - behavioral model
//
// Author: Jeff Link
//*****************************************************************************

module pw_dncntr_b (zeroCnt,value,load,clear,clk);

  output zeroCnt;
  reg    zeroCnt;
  reg    [5:0] count;
  input  [5:0] value;
  input  load,clear,clk;

  initial
    count=0;

  always begin
    if (clear)
      count=0;
    if (zeroCnt & ~load)
      count=0;
    zeroCnt=(count==0);
    #1;
  end

  always @(posedge clk) begin
    #3
    if (clear)
      count=0;
    if (load&~clear)
      count=value;
    if (~load&~clear&~zeroCnt)
      count=count-1;
  end

endmodule
```

Figure 85. Pulse Width Down Counter Behavioral model Verilog® source code.

```
//*****************************************************************************
// File: pw_dncntr_s.v
//
// Description:  Pulse Width Down Counter - structural model
//
// Author: Jeff Link
//*****************************************************************************

module pw_dncntr_s (npwZero,value,load,clear,clk);

  output npwZero;
  wire   [5:0] q,nq;
  input  [5:0] value;
  input  load,clear,clk;
  reg    hi;

  nor #2  nr3(nr3out,q[2],nd2out),
          nr5(nr5out,q[4],nd4out);

  nand #2 nd2(nd2out,nq[1],nq[0]),
          nd4(nd4out,nq[3],nr3out);

  xnor #2 xn1(xn1out,nq[0],nq[1]),
          xn2(xn2out,nd2out,q[2]),
          xn3(xn3out,nr3out,nq[3]),
          xn4(xn4out,nd4out,q[4]),
          xn5(xn5out,nr5out,nq[5]);

  tgmux_b tm0(tm0out,nq[0], value[0],load),
          tm1(tm1out,xn1out,value[1],load),
          tm2(tm2out,xn2out,value[2],load),
          tm3(tm3out,xn3out,value[3],load),
          tm4(tm4out,xn4out,value[4],load),
          tm5(tm5out,xn5out,value[5],load);

  dff_b   dc0(q[0],nq[0],tm0out,hi,dClr,clk),    // count registers
          dc1(q[1],nq[1],tm1out,hi,dClr,clk),
          dc2(q[2],nq[2],tm2out,hi,dClr,clk),
          dc3(q[3],nq[3],tm3out,hi,dClr,clk),
          dc4(q[4],nq[4],tm4out,hi,dClr,clk),
          dc5(q[5],nq[5],tm5out,hi,dClr,clk);

  nor #2  nr0(nr0out,q[0],q[1],q[2]),
          nr1(nr1out,q[3],q[4],q[5]),
          nr2(nr2out,load,npwZero),
          nr3(dClr,  nr2out,clear);

  nand #2 nd0(npwZero,nr0out,nr1out);

  initial
    hi=1;

  always @ (q) begin
    $display("time %0d  \tcount is %b",$time,q);
  end

endmodule
```

Figure 86. Pulse Width Down Counter Structural model Verilog® source code.

4. Repetition Period Down Counter

```
//*********************************************************************
// File: rp_dncntr_test.v
//
// Description:  Test bench for Repetition Period Down Counter
//
// Author: Jeff Link
//*********************************************************************

module rp_dncntr_test;

  reg  [5:0] value;
  reg  load,clear;

  clock    clk1 (clk);
//  rp_dncntr_b down (nrpGT1,value,load,clear,clk);
  rp_dncntr_s down (nrpGT1,value,load,clear,clk);

  initial  begin
    value=99;
    load=0;
    clear=1;
    $display("\t\t\t  nrpGT1 clear load");
    $monitor("time %0d  \t    %b     %b     %b",$time,nrpGT1,clear,load);
    #25
    load=1;
    #40
    clear=0;  // 65
    #100
    load=0;  // 165
    #100
    load=1;  // 265
    #80
    load=0;  // 345
    #5000     // 745
    $finish;
  end
endmodule
```

Figure 87. Repetition Period Down Counter Test Bench Verilog® source code.

```
//*********************************************************************
// File: rp_dncntr_b.v
//
// Description:  Repetition Period Down Counter - behavioral model
//
// Author: Jeff Link
//*********************************************************************

module rp_dncntr_b (rpGT1,value,load,clear,clk);

  output rpGT1;
  reg    rpGT1,zeroCnt;
  reg    [7:0] count;
  input  [5:0] value;
  input  load,clear,clk;

  initial
    count=0;

  always begin
    if (clear)
      count=0;
    if (zeroCnt & ~load)
      count=0;
    zeroCnt=(count==0);
    rpGT1  =(count>1);
    #1;
  end

  always @(posedge clk) begin
    #3
    if (clear)
      count=0;
    if (load&~clear)
      count={value,2'b00};
    if (~load&~clear&~zeroCnt)
      count=count-1;
  end

endmodule
```

Figure 88. Repetition Period Down Counter Behavioral model Verilog® source code.

```
//*********************************************************************
// File: rp_dncntr_s.v
//
// Description:  Repetition Period Down Counter - structural model
//
// Author: Jeff Link
//*********************************************************************

module rp_dncntr_s (nrpGT1,value,load,clear,clk);
  output nrpGT1;
```

Figure 89. Repetition Period Down Counter Structural model Verilog® source code.

```
  wire   [7:0] q,nq;
  input  [5:0] value;
  input  load,clear,clk;
  reg    hi,lo;

  nand #2 nd2(nd2out,nq[1],nq[0]),
          nd4(nd4out,nq[3],nr3out),
          nd6(nd6out,nq[5],nr5out);

  nor #2  nr3(nr3out,q[2],nd2out),
          nr5(nr5out,q[4],nd4out),
          nr7(nr7out,q[6],nd6out);

  xnor #2 xn1(xn1out,nq[0],nq[1]),
          xn2(xn2out,nd2out,q[2]),
          xn3(xn3out,nr3out,nq[3]),
          xn4(xn4out,nd4out,q[4]),
          xn5(xn5out,nr5out,nq[5]),
          xn6(xn6out,nd6out,q[6]),
          xn7(xn7out,nr7out,nq[7]);

  tgmux_b tm0(tm0out,nq[0], lo,      load),
          tm1(tm1out,xn1out,lo,      load),
          tm2(tm2out,xn2out,value[0],load),
          tm3(tm3out,xn3out,value[1],load),
          tm4(tm4out,xn4out,value[2],load),
          tm5(tm5out,xn5out,value[3],load),
          tm6(tm6out,xn6out,value[4],load),
          tm7(tm7out,xn7out,value[5],load);

  dff_b   dc0(q[0],nq[0],tm0out,hi,dClr,clk),    // count registers
          dc1(q[1],nq[1],tm1out,hi,dClr,clk),
          dc2(q[2],nq[2],tm2out,hi,dClr,clk),
          dc3(q[3],nq[3],tm3out,hi,dClr,clk),
          dc4(q[4],nq[4],tm4out,hi,dClr,clk),
          dc5(q[5],nq[5],tm5out,hi,dClr,clk),
          dc6(q[6],nq[6],tm6out,hi,dClr,clk),
          dc7(q[7],nq[7],tm7out,hi,dClr,clk);

  nor #2  nr0(nr0out,q[1],q[2]),
          nr1(nr1out,q[3],q[4]),
          nr2(nr2out,q[5],q[6],q[7]),
          nr4(nr4out,load,ZCnt),
          nr6(dClr,  nr4out,clear);

  nand #2 nd0(ZCnt,nq[0],nrpGT1);

  and #2  an0(nrpGT1,nr0out,nr1out,nr2out);

  initial begin
    hi=1;
    lo=0;
  end

  always @(q) begin
    $display("time %0d  \tcount is %b",$time,q);
  end
endmodule
```

Figure 89. Repetition Period Down Counter Structural model Verilog® source code. (continued)

5. Clock Divider

```
//*********************************************************************
// File: clk_div_test.v
//
// Description:  Test bench for Clock Divider
//
// Author: Jeff Link
//*********************************************************************

module clk_div_test;

  reg     reset;
  integer ii;       // loop counter

  clock #(50) clk0 (clk);
//  clk_div_b    cd0  (fr250, fr62, clk, reset);
  clk_div_s    cd0  (fr250, fr62, clk, reset);

  initial  begin
    $display("          \tfr250 fr62");
    $monitor("time %0d  \t  %b    %b",$time,fr250,fr62);
    reset=1;
    #2
    reset=0;
    ii = 0;
    while (ii<3) begin
      #5;
    end
    $finish;
  end

  always @ (posedge fr62)
    ii=ii+1;

endmodule
```

Figure 90. Clock Divider Test Bench Verilog® source code.

```
//*****************************************************************************
// File: clk_div_b.v
//
// Description:  Clock Divider - behavioral model
//
// Author: Jeff Link
//*****************************************************************************

`define base 64

module clk_div_b (fr250, fr62, clk, reset);
  output fr250, fr62;
  reg    fr250, fr62;
  input  clk, reset;
  integer count1;

  initial begin
    count1=0;
    fr250=0;
    fr62=0;
  end

  always  @(posedge clk)  begin
    if (~reset) begin
      count1 = count1+1;
      if (count1%`base == 0) begin
        fr250 = ~fr250;
      end
      if (count1%(4*`base) == 0) begin
        fr62 = ~fr62;
        count1 =0;
      end
    end
  end

  always  begin
    #1
    if (reset) begin
      count1 =0;
    end
  end
endmodule
```

Figure 91. Clock Divider Behavioral model Verilog® source code.

```
//*****************************************************************************
// File: clk_div_s.v
//
// Description:  Clock Divider - structural model
//
// Author: Jeff Link
//*****************************************************************************


```

Figure 92. Clock Divider Structural model Verilog® source code.

```
module clk_div_s (fr250, fr62, clk, reset);
  output fr250, fr62;
  reg    fr250, fr62;
  wire   [13:0] q,nq;
  input  clk, reset;
  reg    hi;

  not #1  nt0 (nReset,reset);
  dff_b   dc0  (q[0], nq[0], nq[0], hi,nReset,clk),   // count registers
          dc1  (q[1], nq[1], td1in, hi,nReset,clk),
          dc2  (q[2], nq[2], td2in, hi,nReset,clk),
          dc3  (q[3], nq[3], td3in, hi,nReset,clk),
          dc4  (q[4], nq[4], td4in, hi,nReset,clk),
          dc5  (q[5], nq[5], td5in, hi,nReset,clk),
          dc6  (q[6], nq[6], td6in, hi,nReset,clk),
          dc7  (q[7], nq[7], td7in, hi,nReset,clk),
          dc8  (q[8], nq[8], td8in, hi,nReset,clk),
          dc9  (q[9], nq[9], td9in, hi,nReset,clk),
          dc10 (q[10],nq[10],td10in,hi,nReset,clk),
          dc11 (q[11],nq[11],td11in,hi,nReset,clk), // 250 Hz for 1 MHz clk
          dc12 (q[12],nq[12],td12in,hi,nReset,clk),
          dc13 (q[13],nq[13],td13in,hi,nReset,clk);
  xor #2  xr1 (td1in, q[0],   q[1]),
          xr2 (td2in, nd2out, nq[2]),
          xr3 (td3in, nr3out, q[3]),
          xr4 (td4in, nd4out, nq[4]),
          xr5 (td5in, nr5out, q[5]),
          xr6 (td6in, nd6out, nq[6]),
          xr7 (td7in, nr7out, q[7]),
          xr8 (td8in, nd8out, nq[8]),
          xr9 (td9in, nr9out, q[9]),
          xr10(td10in,nd10out,nq[10]),
          xr11(td11in,nr11out,q[11]),
          xr12(td12in,nd12out,nq[12]),
          xr13(td13in,nr13out,q[13]);
  nand #2 nd2 (nd2out, q[1], q[0]),
          nd4 (nd4out, q[3], nr3out),
          nd6 (nd6out, q[5], nr5out),
          nd8 (nd8out, q[7], nr7out),
          nd10(nd10out,q[9], nr9out),
          nd12(nd12out,q[11],nr11out);
  nor #2  nr3 (nr3out, nq[2], nd2out),
          nr5 (nr5out, nq[4], nd4out),
          nr7 (nr7out, nq[6], nd6out),
          nr9 (nr9out, nq[8], nd8out),
          nr11(nr11out,nq[10],nd10out),
          nr13(nr13out,nq[12],nd12out);

  initial
    hi=1;

  always  begin
    fr250 = nq[3];  // used vice 11 for simulation speed
    fr62  = nq[5];
    #1;
  end

endmodule
```

Figure 92. Clock Divider Structural model Verilog® source code. (continued)

E. SUPPORT COMPONENTS

1. Clock with Parametric Half-Period

```
//*********************************************************************
// File: clock_test.v
//
// Description:  Test bench for Clock with Parametric Half-Period
//
// Author: Jeff Link
//*********************************************************************

module dff_test;

  clock        clok1 (clk1);
  clock #(50)  clok2 (clk2);
  clock #(100) clok3 (clk3);

  initial  begin
    $display("\t\t\tclk1 clk2 clk3");
    $monitor("time %0d  \t %b    %b    %b",$time,clk1,clk2,clk3);
    #501
    $finish;
  end
endmodule
```

Figure 93. Clock with Parametric Half-Period Test Bench Verilog® source code.

```
//*********************************************************************
// File: clock.v
//
// Description:  Clock with Parametric Half-Period
//
// Author: Jeff Link
//*********************************************************************

module clock (clk);
  parameter delay=10;
  output  clk;
  reg     clk;
  initial
    clk = 1;
  always
    #(delay) clk = ~clk;
endmodule
```

Figure 94. Clock with Parametric Half-Period Behavioral model Verilog® source code.

2. D flip-flop, positive edge triggered

```
//*********************************************************************
// File: dff_test.v
//
// Description:  Test bench for D flip-flop
//
// Author: Jeff Link
//*********************************************************************

module dff_test;

  reg  d,nP,nC;
  wire clk;

  clock  clk1 (clk);
  dff_b  dff1 (q,nq,nq,nP,nC,clk);

  initial  begin
    d=1;
    nP=1;
    nC=0;
    $display("\t\t\tq nq d nP nC");
    $monitor("time %0d  \t%b %b  %b  %b  %b",$time,q,nq,d,nP,nC);
    #12
    nC=1;
    #20
    d=0;
    #20
    nP=0;
    #20
    nP=1;
    #20
    d=1;
    #20
    nC=0;
    #20
    nC=1;
    #20
    d=0;
    $finish;
  end
endmodule
```

Figure 95. D flip-flop Test Bench Verilog® source code.

```
//*****************************************************************************
// File: dff_b.v
//
// Description:  D flip-flop, positive edge triggered - behavioral model
//
// Author: Jeff Link
//*****************************************************************************

module dff_b (q,nq,d,nP,nC,clk);

  output q,nq;
  reg    q,nq;
  input  d,nP,nC,clk;

  always @(posedge clk) begin
    if (nP&nC) begin
      q  = #3 d;
      nq = ~q;
    end
  end

  always begin
    #1
    if (~nC) begin
      q=0;
      nq=1;
    end
  end

  always begin
    #1
    if (~nP&nC) begin
      q=1;
      nq=0;
    end
  end
endmodule
```

Figure 96. D flip-flop Behavioral model Verilog® source code.

3. Transmission Gate MUX

```
//*********************************************************************
// File: tgmux_test.v
//
// Description:  Test bench for Transmission Gate MUX
//
// Author: Jeff Link
//*********************************************************************

module tgate_test;

  reg  Ain,Bin,select;

  tgmux_b  mux1 (out,Ain,Bin,select);

  initial  begin
    Ain=1;
    Bin=0;
    select=0;
    $display("\t\t\tout Ain Bin sel");
    $monitor("time %0d  \t %b   %b   %b   %b",$time,out,Ain,Bin,select);
    #10 select=1;
    #10 select=0;
    #10
    Ain=0;
    Bin=1;
    #10 select=1;
    #10 select=0;
    #20;
    $finish;
  end
endmodule
```

Figure 97. Transmission Gate MUX Test Bench Verilog® source code.

```
//*********************************************************************
// File: tgmux_b.v
//
// Description:  Transmission Gate MUX - behavioral model
//
// Author: Jeff Link
//*********************************************************************

module tgmux_b (out,Ain,Bin,select);

  output out;
  reg    out;
  input  Ain,Bin,select;

  always
    #1
    if (select)
      out = Bin;
    else
      out = Ain;
endmodule
```

Figure 98. Transmission Gate MUX Behavioral model Verilog® source code.

APPENDIX B. SYSTEM DESIGN SCHEMATICS

This appendix provides the design schematics for the Tactor Interface Chip elements. The schematics were initially designed by hand using circuit examples contained in References 1 and 8. After the circuits were iteratively revised and simulated, the designs were reproduced graphically for reference and documentation. The schematics diagrams are divided into sections based on their parent functional module.

A. SERIAL DATA RECEIVER

1. Twelve-Bit Input Shift Register

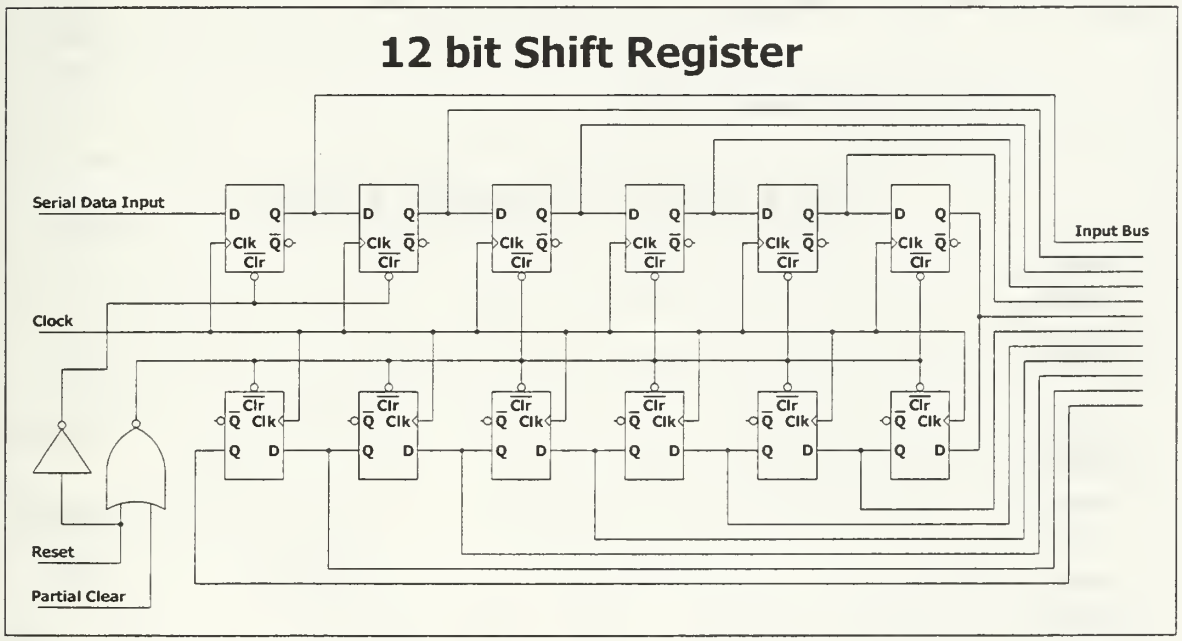


Figure 99. Structural Schematic for the Twelve-Bit Shift Register.

2. Eight-Bit Data Latch

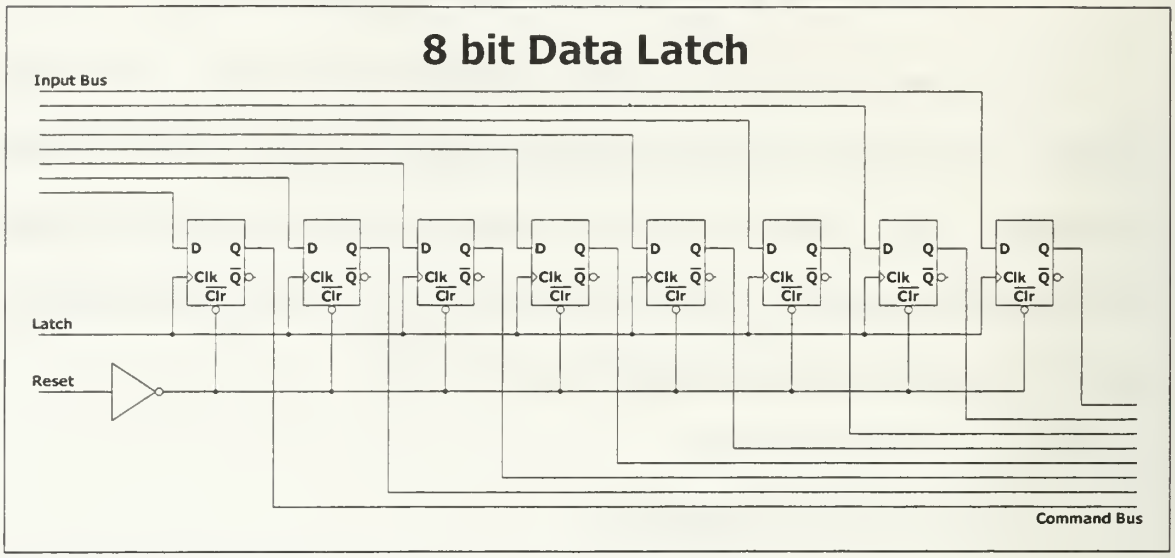


Figure 100. Structural Schematic for the Eight-Bit Data Latch.

3. Input Stream Validity Check

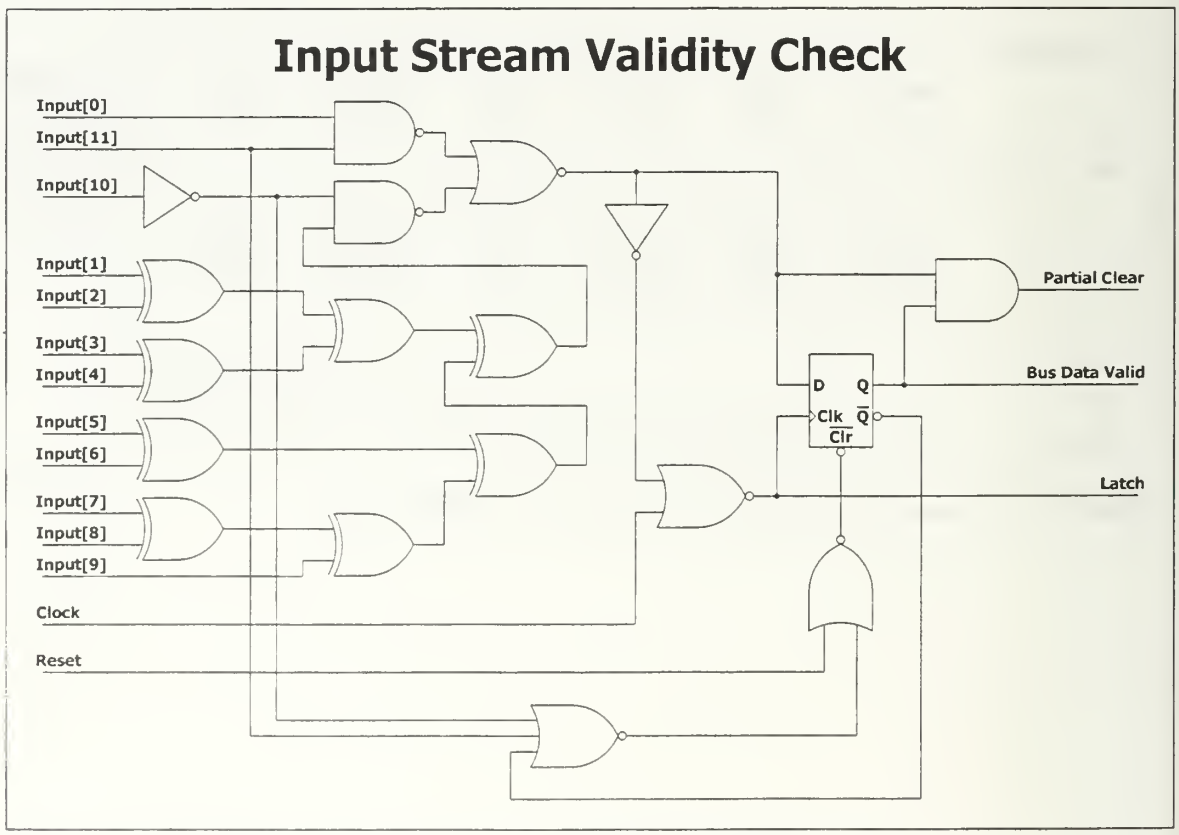


Figure 101. Structural Schematic for the Input Stream Validity Check.

C. COMMAND DECODER AND CONTROLLER

1. Command Sequence Controller

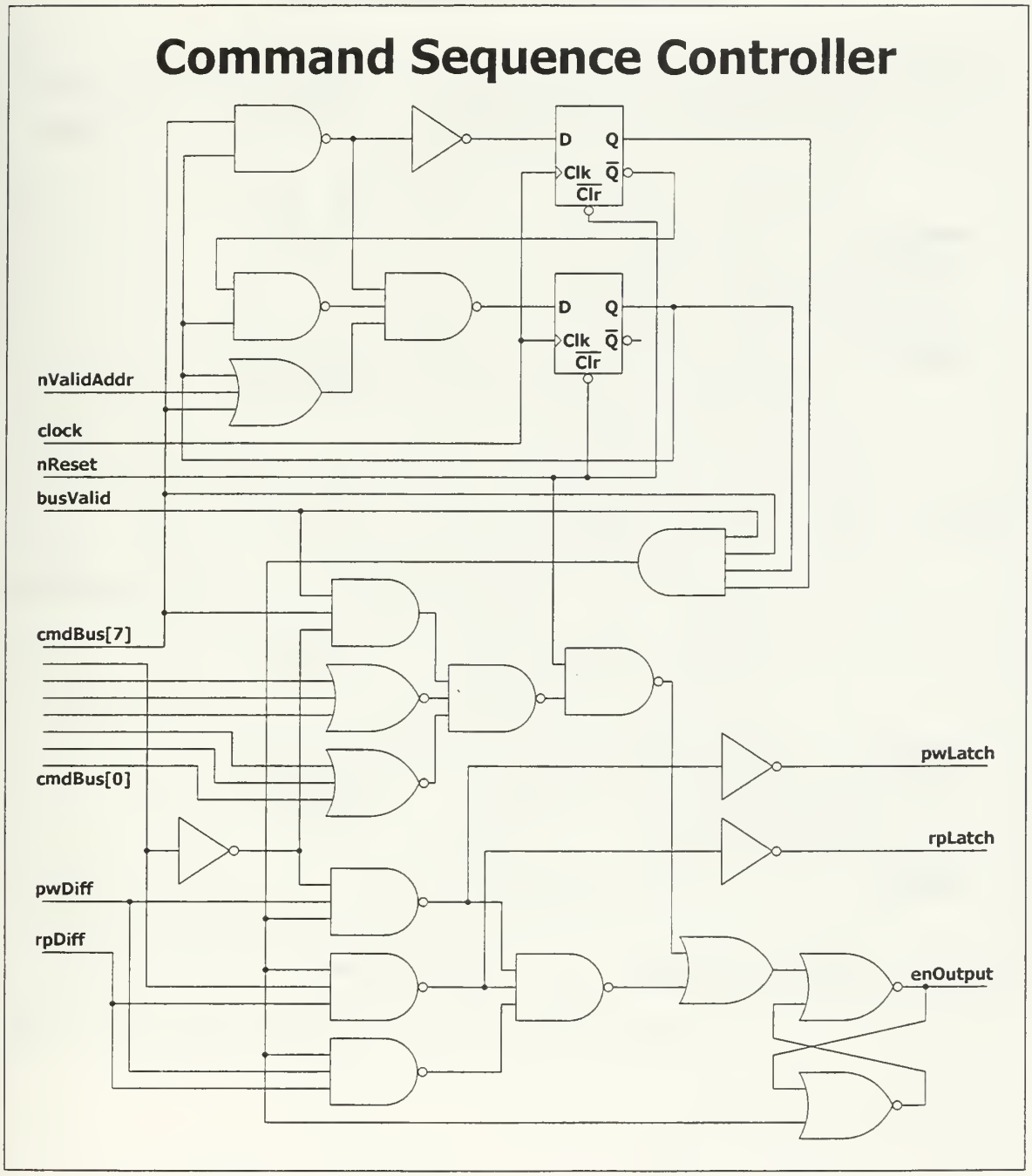


Figure 102. Structural Schematic for the Command Sequence Controller.

2. Address Comparator

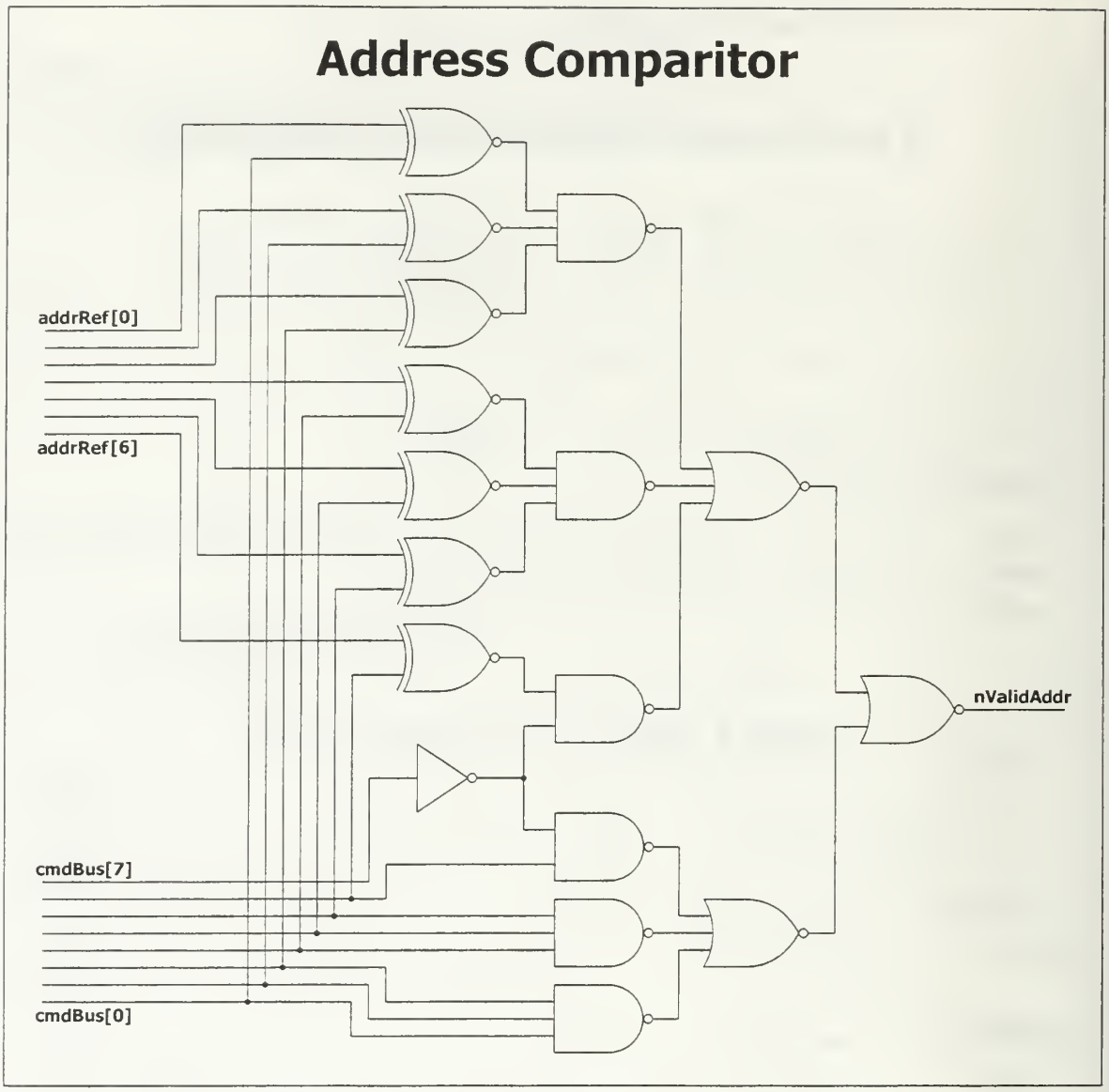


Figure 103. Structural Schematic for the Address Comparator.

3. Pulse Width Register

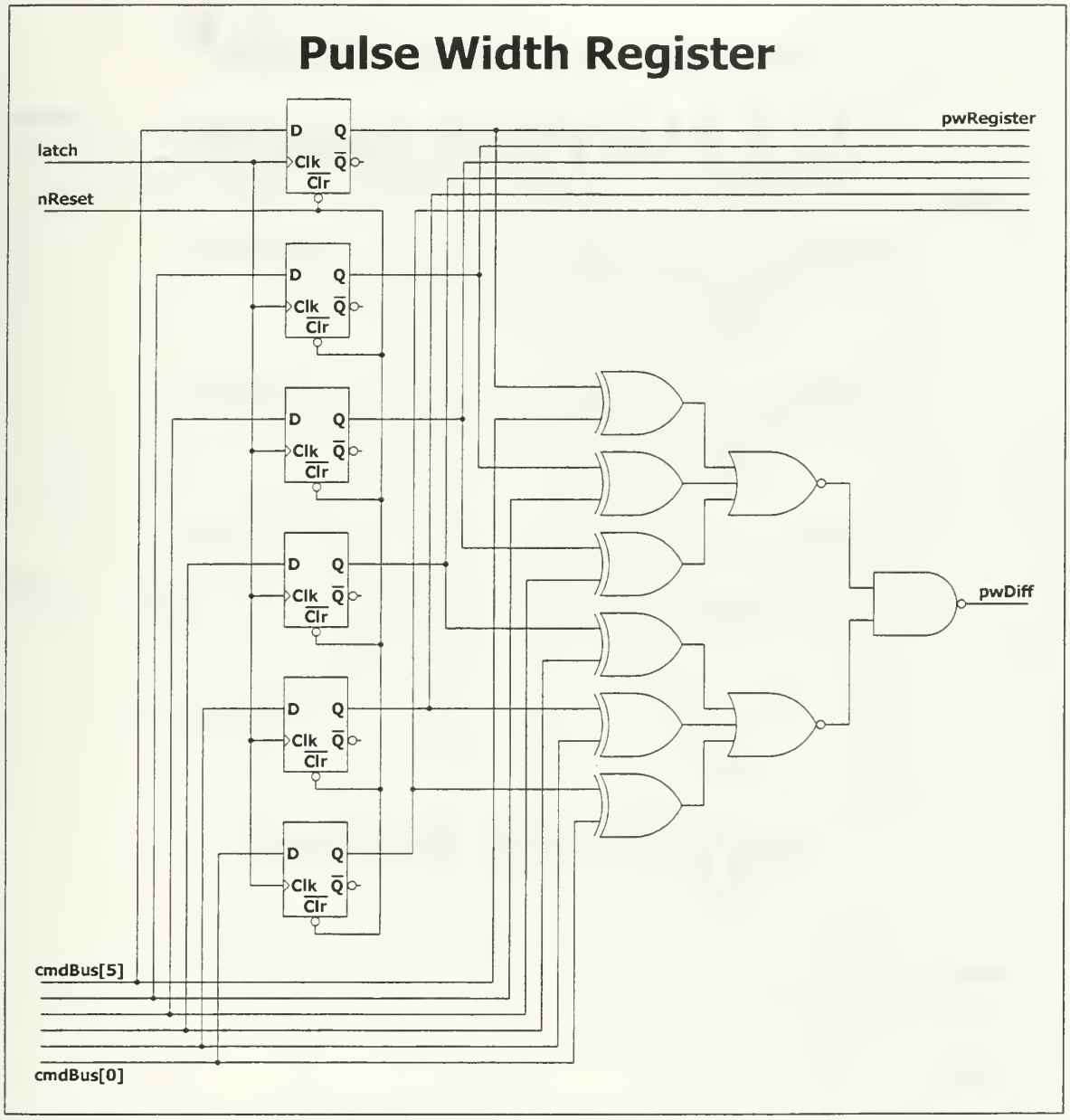


Figure 104. Structural Schematic for the Pulse Width Register.

4. Repetition Period Register

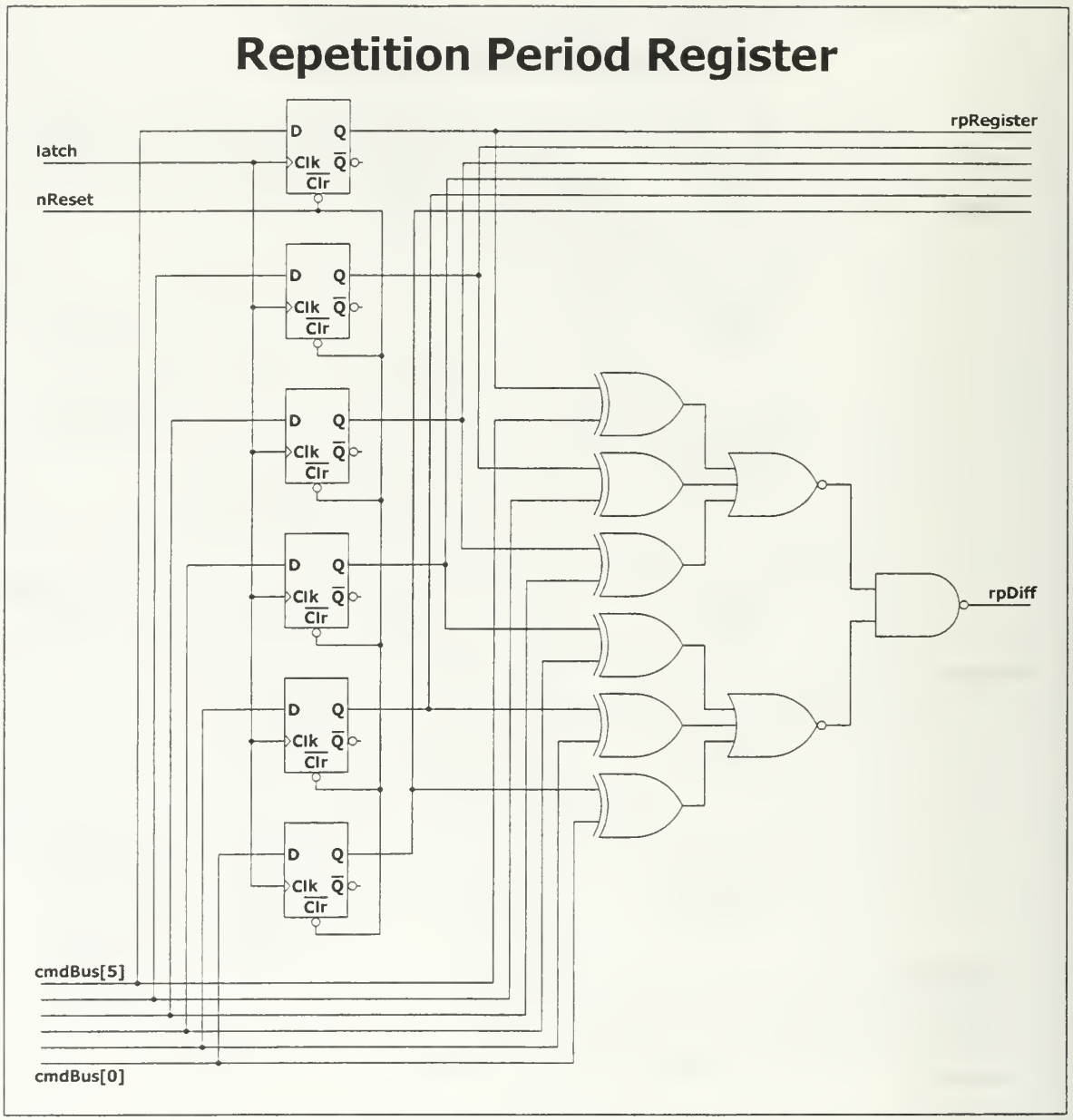


Figure 105. Structural Schematic for the Repetition Period Register.

D. TACTOR POWER CONTROLLER

1. Power Control Logic

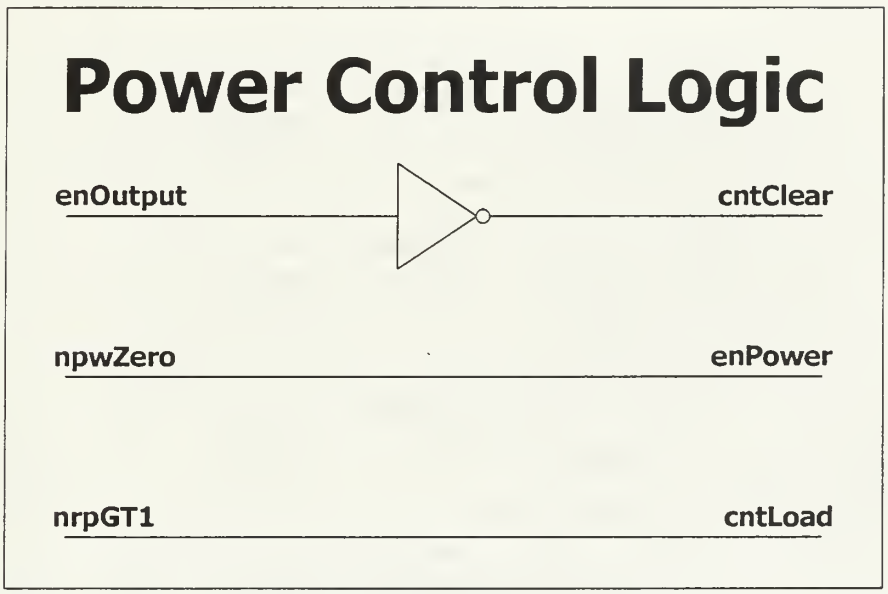


Figure 106. Structural Schematic for the Power Control Logic.

2. Power Oscillator

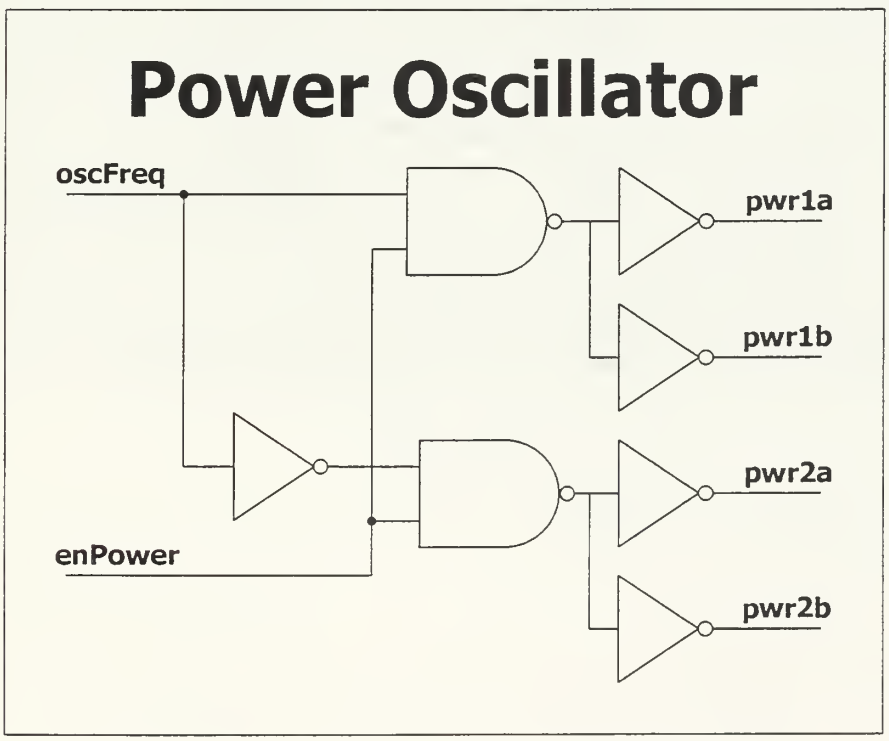


Figure 107. Structural Schematic for the Power Oscillator.

3. Pulse Width Down Counter

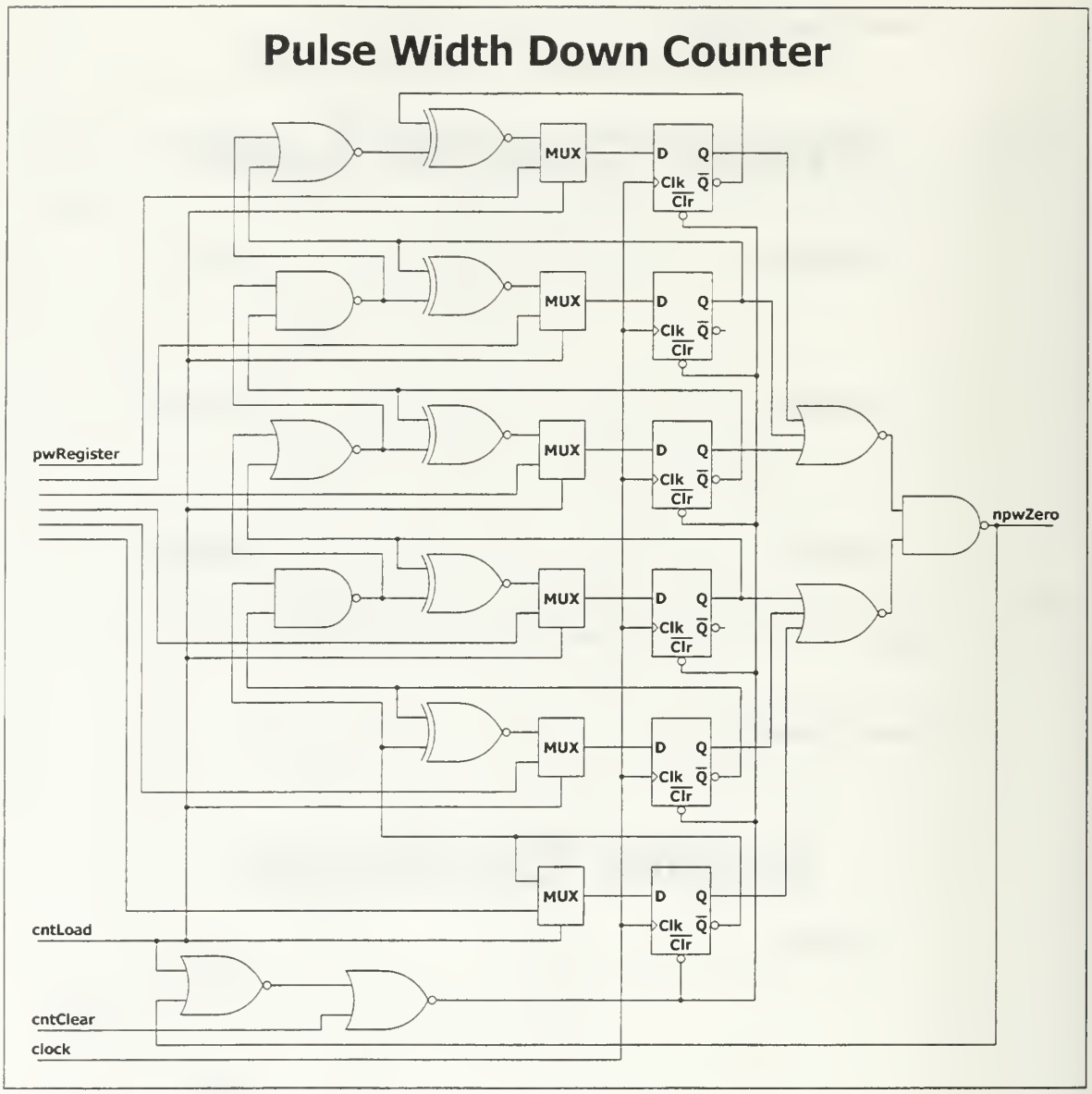


Figure 108. Structural Schematic for the Pulse Width Down Counter.

4. Repetition Period Down Counter

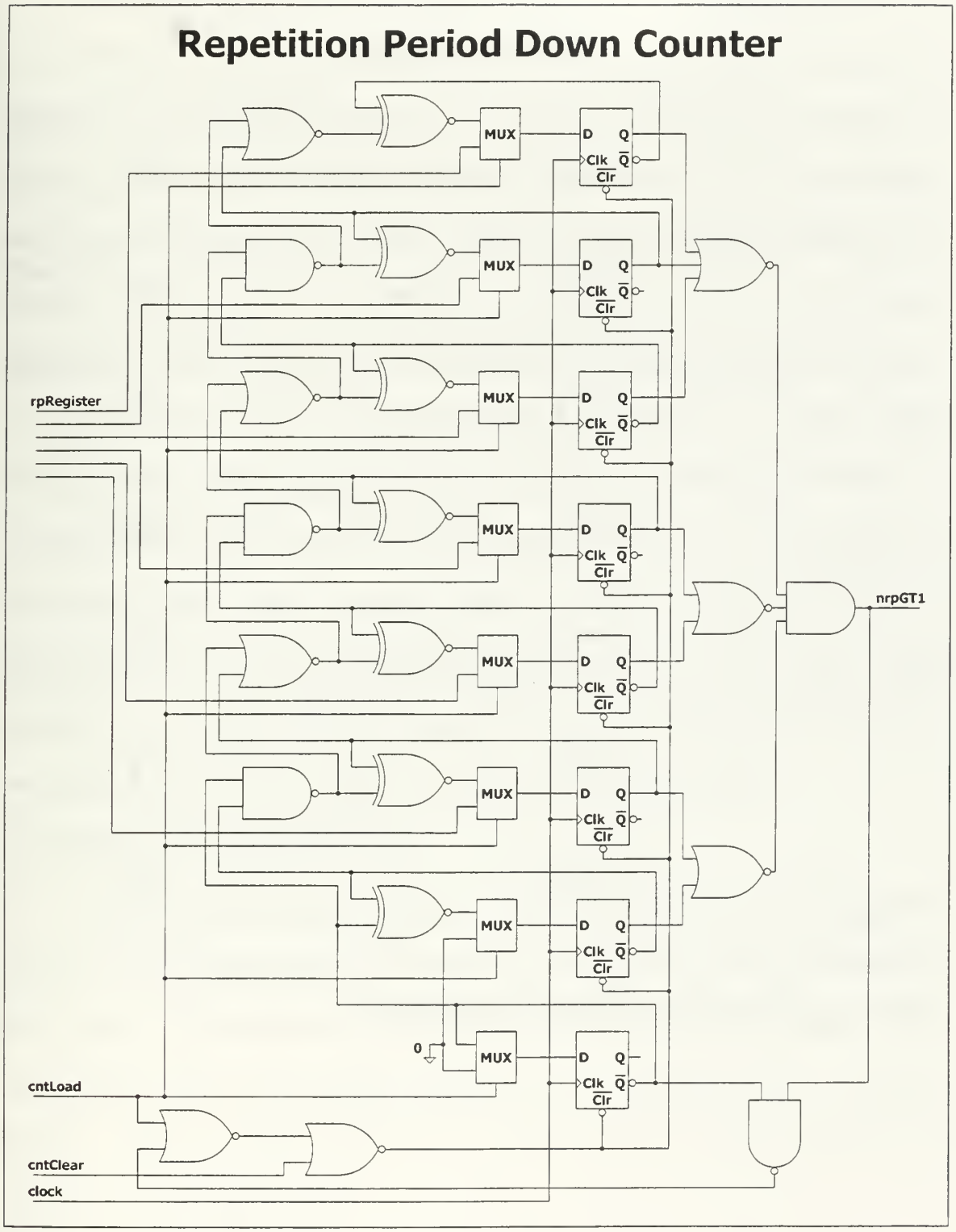


Figure 109. Structural Schematic for the Repetition Period Down Counter.

5. Clock Divider

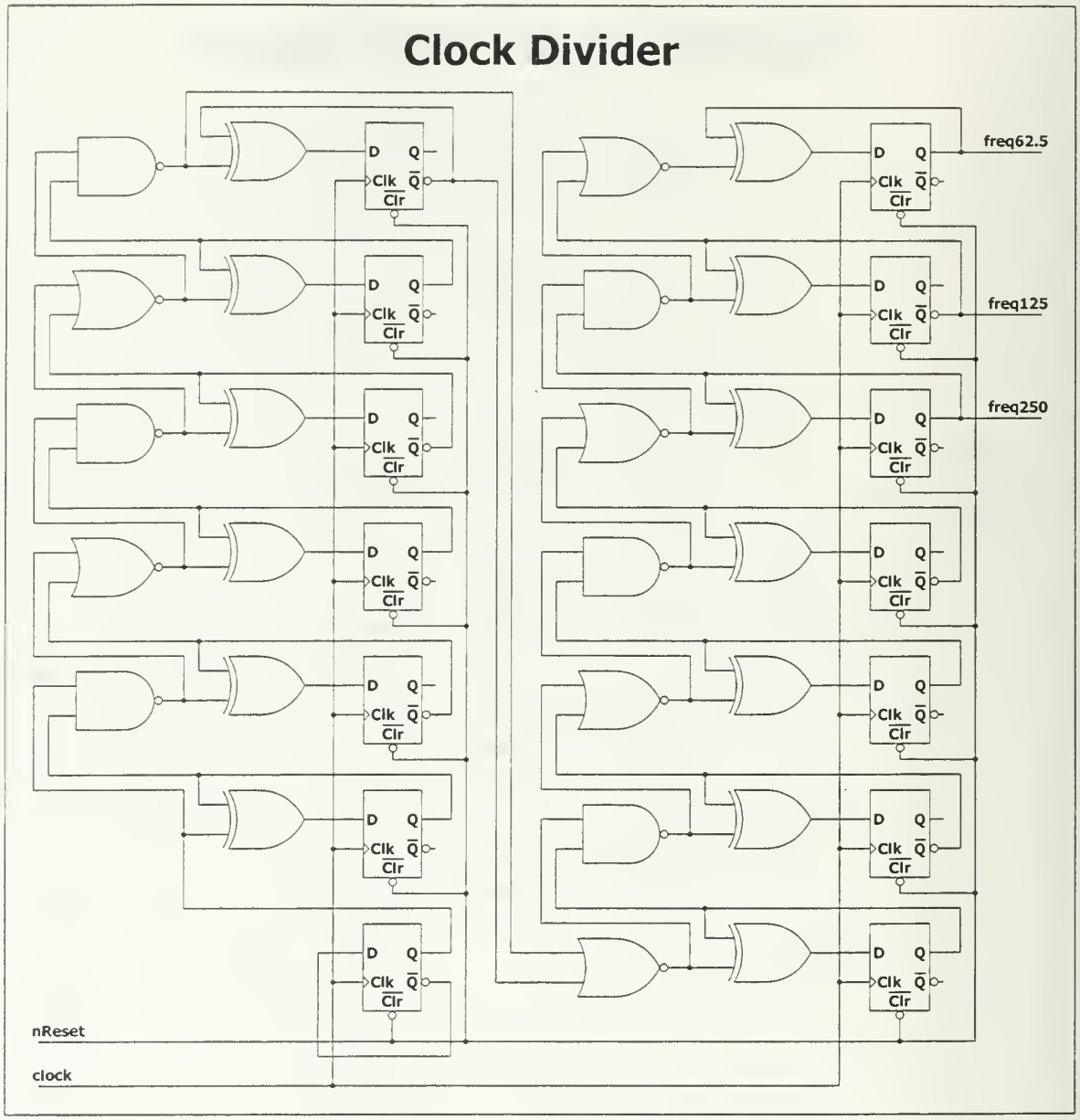


Figure 110. Structural Schematic for the Clock Divider.

APPENDIX C. STRUCTURAL EVALUATION USING SPICE

SPICE is a circuit simulation program developed by Dr. Lawrence Nagel in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkely. The SPICE model for FETs allows defining the semiconductor devices using pertinent parameter values. This capability affords designers the opportunity to accurately simulate circuits for evaluation of response time and power consumption. Reference 7 provides a comprehensive presentation of SPICE commands and conventions.

The structural designs presented in Appendix B were evaluated with SPICE to verify circuit response. This appendix documents the SPICE code used for evaluating TIC component design. Testing was limited to representative inputs since circuit behavior was thoroughly evaluated using the Verilog® models presented in Appendix A. Clocked TIC components were tested at 20 MHz instead of the design speed of 1 MHz to allow a safety margin for system speed and to reduce the SPICE simulation time. A brief discussion of each simulation is included to highlight the critical circuit response points.

A. GENERAL DEFINITION FILES

Modeling the TIC circuits required inclusion of FET parameters in all SPICE source code files. Each logic element must be defined as a compilation of FETs since the circuits were defined using fundamental logic elements rather than individual transistors. These logic element definitions must also be included in all SPICE source code files. These two inclusion requirements are separated into two different files to provide three levels of physical component abstraction.

1. CMOS FET Model Parameters

The CMOS FET model parameters are provided by the expected VLSI chip manufacturer. The parametric values are actually determined using a combination of theoretical response and empirical measurements. These FET definitions are stored in a separate file that must be included in every circuit definition. This separation ensures the latest updated values are automatically used every time a SPICE simulation is executed. A listing of the CMOS FET model definitions is included as Figure 111. Reference 9 contains a detailed description of each of the FET model parameters.

```
* cmos.cir ==>  CMOS PFET & NFET model definitions

* MOSIS PARAMETRIC TEST RESULTS
* DATE:  1 OCTOBER 1997
* RUN: N78K
* VENDOR: ORBIT
* TECHNOLOGY: SCNA20
* FEATURE SIZE: 2.0 MICRONS

.MODEL CMOSN NMOS LEVEL=2 PHI=0.700000 TOX=4.0800E-08 XJ=0.200000U TPG=1
+ VTO=0.8309 DELTA=3.2570E+00 LD=3.2850E-07 KP=6.2842E-05
+ UO=742.5 UEXP=1.9200E-01 UCRIT=2.1830E+04 RSH=6.1490E+00
+ GAMMA=0.5612 NSUB=6.7970E+15 NFS=9.0930E+10 VMAX=5.7540E+04
+ LAMBDA=4.2800E-02 CGDO=4.1704E-10 CGSO=4.1704E-10
+ CGBO=3.4581E-10 CJ=1.2204E-04 MJ=6.3602E-01 CJSW=5.5150E-10
+ MJSW=2.5691E-01 PB=4.4514E-01
* Weff = Wdrawn - Delta_W
* The suggested Delta_W is 2.0000E-09

.MODEL CMOSP PMOS LEVEL=2 PHI=0.700000 TOX=4.0800E-08 XJ=0.200000U TPG=-1
+ VTO=-0.9891 DELTA=1.2110E+00 LD=3.7130E-07 KP=1.7503E-05
+ UO=206.8 UEXP=2.8220E-01 UCRIT=1.1030E+05 RSH=1.0210E-01
+ GAMMA=0.7803 NSUB=1.3140E+16 NFS=7.1500E+11 VMAX=1.2110E+05
+ LAMBDA=5.3880E-02 CGDO=4.7138E-10 CGSO=4.7138E-10
+ CGBO=3.5113E-10 CJ=3.2670E-04 MJ=6.2773E-01 CJSW=3.7671E-10
+ MJSW=1.9873E-01 PB=9.0000E-01
* Weff = Wdrawn - Delta_W
* The suggested Delta_W is 2.3340E-08
```

Figure 111. CMOS PFET and NFET SPICE model definitions.

2. Fundamental Logic Element Definitions

The TIC circuits are defined as combinations of discrete logic elements, thus each element must be defined in terms of the FETs used to implement the logic function. These fundamental logic element definitions are required in every TIC component source file. SPICE provides a convenient method for defining a collection of FETs as an element using the sub-circuit function. The file listing in Figure 112 defines all of the logic elements used in the SPICE simulation source files. This file is imported in every TIC component file using the .INCLUDE command. The CMOS FET definitions are available to all source files since the sub-circuit file includes the CMOS FET definition file as one of its first lines.

```
* subckt.cir ==> CMOS SUB-CIRCUITS for inclusion into other models

* CMOSP & CMOSN model definitions
.INCLUDE cmos.cir

* Inverter Circuit
* define INV - In Out Vdd Gnd
.SUBCKT INV i o v g
Ma v i o v CMOSP W=6U L=2U
Mb o i g g CMOSN W=3U L=2U
.ENDS

* Inverter Circuit - extra large
* define INVx - In Out Vdd Gnd
.SUBCKT INVx i o v g
Ma v i o v CMOSP W=12U L=2U
Mb o i g g CMOSN W=6U L=2U
.ENDS

* Transmission Gate Circuit
* define XGATE - In Out Pgate Ngate Vdd Gnd
.SUBCKT XGATE i o p n v g
Ma i p o v CMOSP W=6U L=2U
Mb i n o g CMOSN W=3U L=2U
.ENDS

* 2-input NAND Circuit
* define NAND2 - Ain Bin Out Vdd Gnd
.SUBCKT NAND2 a b o v g
Ma v a o v CMOSP W=6U L=2U
Mb o a 2 g CMOSN W=3U L=2U
Mc v b o v CMOSP W=6U L=2U
Md 2 b g g CMOSN W=3U L=2U
.ENDS
```

Figure 112. Subcircuits for Fundamental Logic Element SPICE model definitions.

```
* 2-input AND Circuit
* define AND2 - Ain Bin Out Vdd Gnd
.SUBCKT AND2 a b o v g
Xla a b 2 v g NAND2
Xia 2 o v g INV
.ENDS

* 3-input NAND Circuit
* define NAND3 - Ain Bin Cin Out Vdd Gnd
.SUBCKT NAND3 a b c o v g
Ma v a o v CMOSP W=6U L=2U
Mb o a 2 g CMOSN W=3U L=2U
Mc v b o v CMOSP W=6U L=2U
Md 2 b 3 g CMOSN W=3U L=2U
Me v c o v CMOSP W=6U L=2U
Mf 3 c g g CMOSN W=3U L=2U
.ENDS

* 3-input AND Circuit
* define AND3 - Ain Bin Cin Out Vdd Gnd
.SUBCKT AND3 a b c o v g
Xla a b c 2 v g NAND3
Xia 2 o v g INV
.ENDS

* 4-input NAND Circuit
* define NAND4 - Ain Bin Cin Din Out Vdd Gnd
.SUBCKT NAND4 a b c d o v g
Ma v a o v CMOSP W=6U L=2U
Mb o a 2 g CMOSN W=3U L=2U
Mc v b o v CMOSP W=6U L=2U
Md 2 b 3 g CMOSN W=3U L=2U
Me v c o v CMOSP W=6U L=2U
Mf 3 c 4 g CMOSN W=3U L=2U
Mg v d o v CMOSP W=6U L=2U
Mh 4 d g g CMOSN W=3U L=2U
.ENDS

* 4-input AND Circuit
* define AND4 - Ain Bin Cin Din Out Vdd Gnd
.SUBCKT AND4 a b c d o v g
Xla a b c d 2 v g NAND4
Xia 2 o v g INV
.ENDS

* 2-input NOR Circuit
* define NOR2 - Ain Bin Out Vdd Gnd
.SUBCKT NOR2 a b o v g
Ma v a 2 v CMOSP W=6U L=2U
Mb o a g g CMOSN W=3U L=2U
Mc 2 b o v CMOSP W=6U L=2U
Md o b g g CMOSN W=3U L=2U
.ENDS
```

Figure 112. Subcircuits for Fundamental Logic Element SPICE model definitions. (continued)

```
* 2-input OR Circuit
* define OR2 - Ain Bin Out Vdd Gnd
.SUBCKT OR2 a b o v g
Xla a b 2 v g NOR2
Xia 2 o v g INV
.ENDS

* 3-input NOR Circuit
* define NOR3 - Ain Bin Cin Out Vdd Gnd
.SUBCKT NOR3 a b c o v g
Ma v a 2 v CMOSP W=6U L=2U
Mb o a g g CMOSN W=3U L=2U
Mc 2 b 3 v CMOSP W=6U L=2U
Md o b g g CMOSN W=3U L=2U
Me 3 c o v CMOSP W=6U L=2U
Mf o c g g CMOSN W=3U L=2U
.ENDS

* 3-input OR Circuit
* define OR3 - Ain Bin Cin Out Vdd Gnd
.SUBCKT OR3 a b c o v g
Xla a b c 2 v g NOR3
Xia 2 o v g INV
.ENDS

* D-Flip/Flop using transmission gates
* define DFLOPG - Din CLKin Qout nQout Vdd Gnd
.SUBCKT DFLOPG d clk q nq v g
Ma v 3 6 v CMOSP W=6U L=2U
Mb 7 3 g g CMOSN W=3U L=2U
Mc 6 5 2 v CMOSP W=6U L=2U
Md 2 clk 7 g CMOSN W=3U L=2U
Xga d 2 clk 5 v g XGATE
Xgb 3 4 5 clk v g XGATE
Xgc 4 nq clk 5 v g XGATE
Xia q nq v g INV
Xib clk 5 v g INV
Xic 2 3 v g INV
Xid 4 q v g INV
.ENDS
```

Figure 112. Subcircuits for Fundamental Logic Element SPICE model definitions. (continued)

```
* Gated D-Flip/Flop w/ nClear using transmission gates
* define DFLOPGC - Din CLKin nCin Qout nQout Vdd Gnd
.SUBCKT DFLOPGC d clk nc q nq v g
Ma v 2 3 v CMOSP W=6U L=2U
Mb 3 2 9 g CMOSN W=3U L=2U
Me v nc 3 v CMOSP W=6U L=2U
Mf 9 nc g g CMOSN W=3U L=2U
Mg v 2 10 v CMOSP W=6U L=2U
Mh 11 3 g g CMOSN W=3U L=2U
Mi 10 5 2 v CMOSP W=6U L=2U
Mj 2 clk 11 g CMOSN W=3U L=2U
Mk 12 4 q v CMOSP W=6U L=2U
Ml q 4 g g CMOSN W=3U L=2U
Mo v 6 12 v CMOSP W=6U L=2U
Mp q 6 g g CMOSN W=3U L=2U
Xga d 2 clk 5 v g XGATE
Xgb 3 4 5 clk v g XGATE
Xgc 4 nq clk 5 v g XGATE
Xia q nq v g INV
Xib clk 5 v g INV
Xic nc 6 v g INV
.ENDS

* Gated D-Flip/Flop w/ nClear & nPreset using transmission gates
* define DFLOPGCP - Din CLKin nCin nPin Qout nQout Vdd Gnd
.SUBCKT DFLOPGCP d clk nc np q nq v g
Ma 8 2 3 v CMOSP W=6U L=2U
Mb 3 2 9 g CMOSN W=3U L=2U
Mc v 7 8 v CMOSP W=6U L=2U
Md 3 7 9 g CMOSN W=3U L=2U
Me v nc 3 v CMOSP W=6U L=2U
Mf 9 nc g g CMOSN W=3U L=2U
Mg v 2 10 v CMOSP W=6U L=2U
Mh 11 3 g g CMOSN W=3U L=2U
Mi 10 5 2 v CMOSP W=6U L=2U
Mj 2 clk 11 g CMOSN W=3U L=2U
Mk 12 4 q v CMOSP W=6U L=2U
Ml q 4 13 g CMOSN W=3U L=2U
Mm 12 np q v CMOSP W=6U L=2U
Mn 13 np g g CMOSN W=3U L=2U
Mo v 6 12 v CMOSP W=6U L=2U
Mp q 6 g g CMOSN W=3U L=2U
Xga d 2 clk 5 v g XGATE
Xgb 3 4 5 clk v g XGATE
Xgc 4 nq clk 5 v g XGATE
Xia q nq v g INV
Xib clk 5 v g INV
Xic nc 6 v g INV
Xid np 7 v g INV
.ENDS

* Toggle Flip/Flop w/ nClear using transmission gates
* define TFLOPC - Tin CLKin nCin Qout nQout Vdd Gnd
.SUBCKT TFLOPC t clk nc q nq v g
Xla t q 2 v g XOR2
Xda 2 clk nc q nq v g DFLOPC
.ENDS
```

Figure 112. Subcircuits for Fundamental Logic Element SPICE model definitions. (continued)

```
* 2-input XOR Circuit
* define XOR2 - Ain Bin Out Vdd Gnd
.SUBCKT XOR2 a b o v g
Ma v a 2 v CMOSP W=6U L=2U
Mb 2 a g g CMOSN W=3U L=2U
Mc v a 4 v CMOSP W=6U L=2U
Md 4 3 o v CMOSP W=6U L=2U
Me o b 6 g CMOSN W=3U L=2U
Mf 6 a g g CMOSN W=3U L=2U
Mg v b 5 v CMOSP W=6U L=2U
Mh 5 2 o v CMOSP W=6U L=2U
Mi o 3 7 g CMOSN W=3U L=2U
Mj 7 2 g g CMOSN W=3U L=2U
Mk v b 3 v CMOSP W=6U L=2U
Ml 3 b g g CMOSN W=3U L=2U
.ENDS

* 2-input XNOR Circuit
* define XNOR2 - Ain Bin Out Vdd Gnd
.SUBCKT XNOR2 a b o v g
Ma v a 2 v CMOSP W=6U L=2U
Mb 2 a g g CMOSN W=3U L=2U
Mc v a 4 v CMOSP W=6U L=2U
Md 4 b o v CMOSP W=6U L=2U
Me o 3 6 g CMOSN W=3U L=2U
Mf 6 a g g CMOSN W=3U L=2U
Mg v 2 5 v CMOSP W=6U L=2U
Mh 5 3 o v CMOSP W=6U L=2U
Mi o 2 7 g CMOSN W=3U L=2U
Mj 7 b g g CMOSN W=3U L=2U
Mk v b 3 v CMOSP W=6U L=2U
Ml 3 b g g CMOSN W=3U L=2U
.ENDS

* 2-input MUX Circuit
* define MUX - Ain Bin Sel Out Vdd Gnd
.SUBCKT MUX a b s o v g
Ma v s 2 v CMOSP W=6U L=2U
Mb 2 s g g CMOSN W=3U L=2U
Mc a s o v CMOSP W=6U L=2U
Md a 2 o g CMOSN W=3U L=2U
Me b 2 o v CMOSP W=6U L=2U
Mf b s o g CMOSN W=3U L=2U
.ENDS
```

Figure 112. Subcircuits for Fundamental Logic Element SPICE model definitions. (continued)

B. SERIAL DATA RECEIVER

The SPICE model for the Serial Data Receiver is a combination of the models for its subordinate components. A full source code listing is provided in Figure 113. The Serial Data Receiver response is shown in Figure 114. The command packet "0 0 1 1 1 1 1 1 1 0 1" is simulated into the serial data input for the receiver. When the last bit of the valid command is received, the receiver latches the command onto the command bus and provides a bus data valid signal to the other TIC modules. The bus data valid signal is held for ten clock cycles and it is then cleared in preparation for the next possible command packet. The command bus values are not changed until another command is received or until the system is reset. The clearing action at 1.8 μS in Figure 114 is caused by a system reset signal inserted to verify the Serial Data Receiver reset response.

```
* SerialDataReceiver.cir ==> Serial Data Receiver Transient Characteristics

* Logic Gate model definitions
.INCLUDE subckt.cir

* Power Supplies
VDS 1 0 5

* Input Signals
Vd   d   0 PWL(0 5 299.5n 5 301.5n 0 399.5n 0 401.5n 5 749.5n 5 751.5n 0
799.5n 0 801.5n 5 1 5)
Vrst rst 0 PWL(0 0 1795n 0 1796n 5 1799n 5 1800n 0)
Vclk clk 0 PULSE(0 5 24.5N 1N 1N 24N 50N)

* Twelve-bit shift register
Xsd0 d clk nR i0 ni0 1 0 DFLOPGC
Xsd1 i0 clk nR i1 ni1 1 0 DFLOPGC
Xsd2 i1 clk npC i2 ni2 1 0 DFLOPGC
Xsd3 i2 clk npC i3 ni3 1 0 DFLOPGC
Xsd4 i3 clk npC i4 ni4 1 0 DFLOPGC
Xsd5 i4 clk npC i5 ni5 1 0 DFLOPGC
Xsd6 i5 clk npC i6 ni6 1 0 DFLOPGC
Xsd7 i6 clk npC i7 ni7 1 0 DFLOPGC
Xsd8 i7 clk npC i8 ni8 1 0 DFLOPGC
Xsd9 i8 clk npC i9 ni9 1 0 DFLOPGC
Xsd10 i9 clk npC i10 ni10 1 0 DFLOPGC
Xsd11 i10 clk npC i11 ni11 1 0 DFLOPGC
Xsi rst nR 1 0 INV
Xsnr rst pC npC 1 0 NOR2
```

Figure 113. Serial Data Receiver SPICE model source code.

```
* Eight-bit data latch
Xld0 i2 latch nR q0 nq0 1 0 DFLOPGC
Xld1 i3 latch nR q1 nq1 1 0 DFLOPGC
Xld2 i4 latch nR q2 nq2 1 0 DFLOPGC
Xld3 i5 latch nR q3 nq3 1 0 DFLOPGC
Xld4 i6 latch nR q4 nq4 1 0 DFLOPGC
Xld5 i7 latch nR q5 nq5 1 0 DFLOPGC
Xld6 i8 latch nR q6 nq6 1 0 DFLOPGC
Xld7 i9 latch nR q7 nq7 1 0 DFLOPGC

* Input stream validity check
Xvx0 i1 i2 xo0 1 0 XOR2
Xvx1 i3 i4 xo1 1 0 XOR2
Xvx2 i5 i6 xo2 1 0 XOR2
Xvx3 i7 i8 xo3 1 0 XOR2
Xvx4 i9 xo3 xo4 1 0 XOR2
Xvx5 xo0 xo1 xo5 1 0 XOR2
Xvx6 xo2 xo4 xo6 1 0 XOR2
Xvx7 xo5 xo6 xo7 1 0 XOR2
Xva0 i0 i11 ao0 1 0 NAND2
Xva1 ni10 xo7 ao1 1 0 NAND2
Xvn0 ao0 ao1 frm 1 0 NOR2
Xvi0 frm nfrm 1 0 INV
Xvn1 nbdv ni10 i11 no1 1 0 NOR3
Xvn2 nfrm clk latch 1 0 NOR2
Xvn3 rst no1 no3 1 0 NOR2
Xvd0 frm latch no3 bdv nbdv 1 0 DFLOPGC
Xva2 bdv frm pC 1 0 AND2

* Simulation Parameters
.TRAN .1N 2000N 0 1n

.END
```

Figure 113. Serial Data Receiver SPICE model source code. (continued)

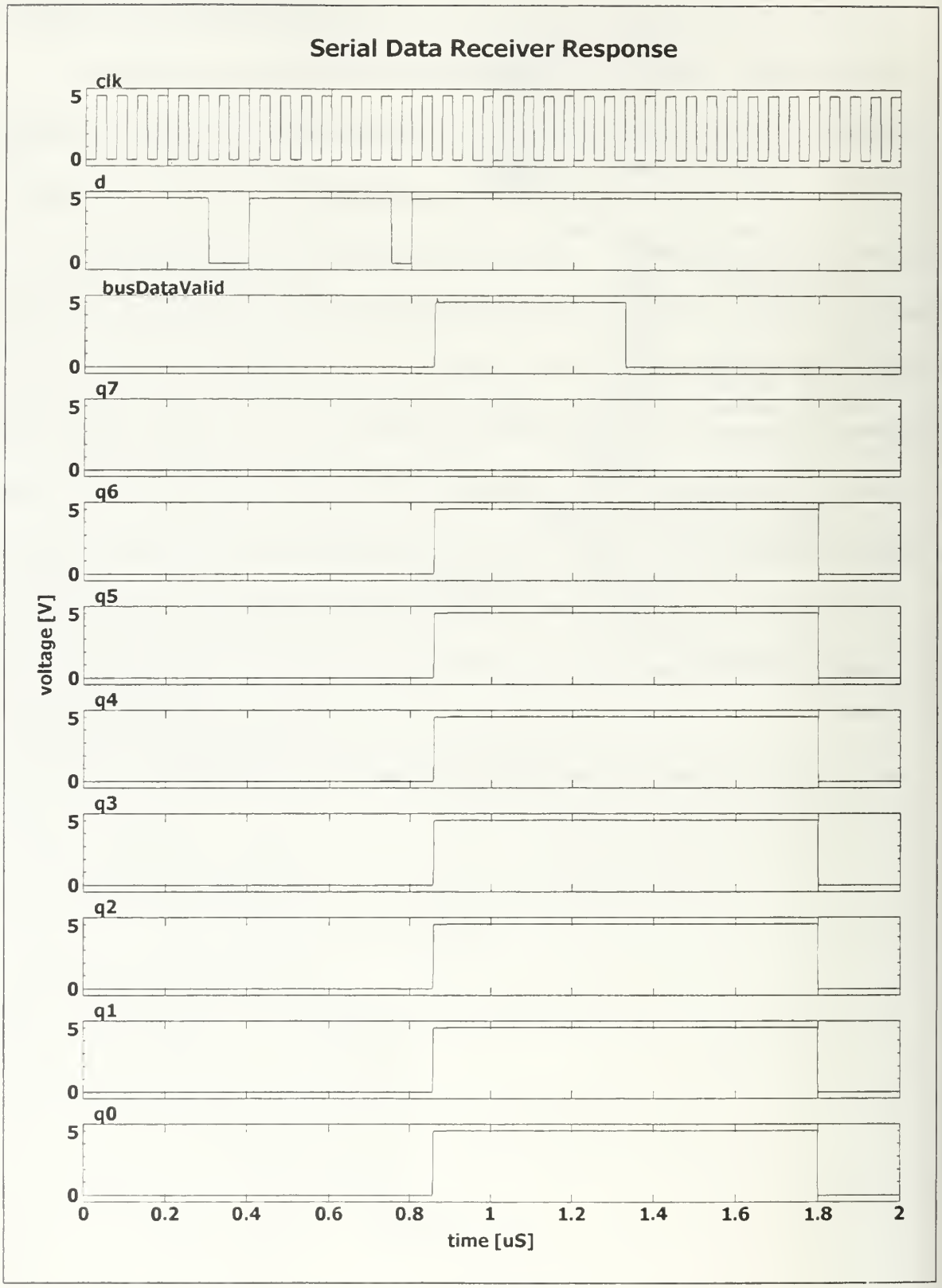


Figure 114. Serial Data Receiver SPICE model response.

1. Twelve-Bit Input Shift Register

The SPICE model for the twelve-bit input shift register implements the structural design of Figure 99. A full source code listing is provided in Figure 115. The command packet "0 0 1 1 1 1 1 1 1 0 1" is simulated into the serial data input for the shift register. The shift register response is shown in Figure 116, which illustrates the input bits shift one position at every clock cycle. The clearing action at 1.7 μS in Figure 116 is caused by a partial-clear signal; demonstrating that only the highest ten bits are cleared. The clearing action at 1.8 μS is caused by a reset signal; demonstrating that all bits are cleared for a system reset.

```
* shift12.cir ==>  Twelve-Bit Shift Register Transient Characteristics

* Logic Gate model definitions
.INCLUDE subckt.cir

* Power Supplies
VDS 1 0 5

* Input Signals
Vd   d   0 PWL(0 5 299.5n 5 301.5n 0 399.5n 0 401.5n 5 749.5n 5 751.5n 0
               799.5n 0 801.5n 5 1 5)
Vrst rst 0 PWL(0 0 1795n 0 1796n 5 1799n 5 1800n 0)
Vpc  pC  0 PWL(0 0 1695n 0 1696n 5 1699n 5 1700n 0)
Vclk clk 0 PULSE(0 5 24.5n 1n 1n 24n 50n)

* Twelve-bit shift register
Xd0 d clk nR q0 nq0 1 0 DFLOPGC
Xd1 q0 clk nR q1 nq1 1 0 DFLOPGC
Xd2 q1 clk npC q2 nq2 1 0 DFLOPGC
Xd3 q2 clk npC q3 nq3 1 0 DFLOPGC
Xd4 q3 clk npC q4 nq4 1 0 DFLOPGC
Xd5 q4 clk npC q5 nq5 1 0 DFLOPGC
Xd6 q5 clk npC q6 nq6 1 0 DFLOPGC
Xd7 q6 clk npC q7 nq7 1 0 DFLOPGC
Xd8 q7 clk npC q8 nq8 1 0 DFLOPGC
Xd9 q8 clk npC q9 nq9 1 0 DFLOPGC
Xd10 q9 clk npC q10 nq10 1 0 DFLOPGC
Xd11 q10 clk npC q11 nq11 1 0 DFLOPGC
Xi rst nR 1 0 INV
Xnr rst pC npC 1 0 NOR2

* Simulation Parameters
.TRAN .1n 2000n 0 1n

.END
```

Figure 115. Twelve-Bit Input Shift Register SPICE model source code.

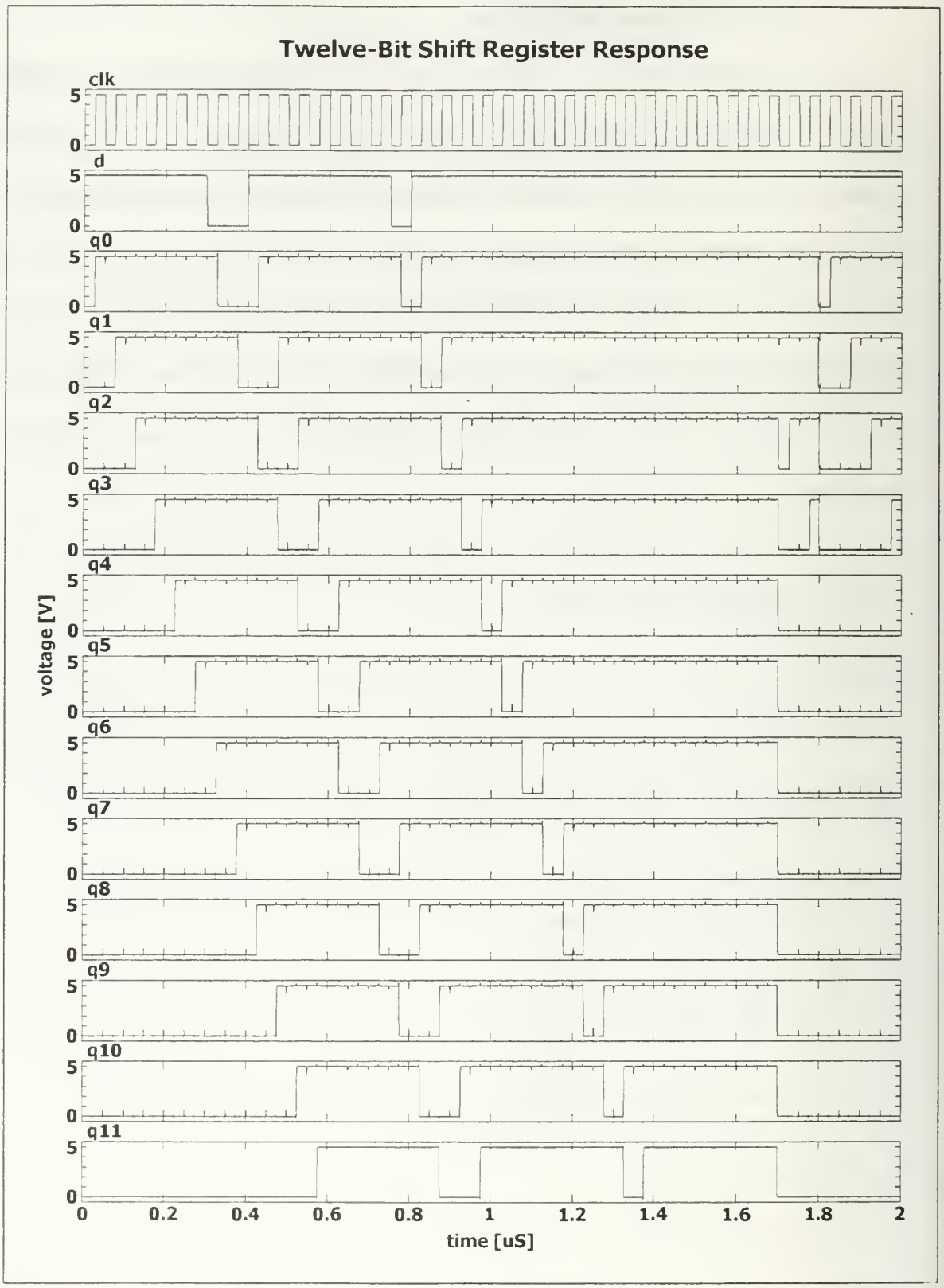


Figure 116. Twelve-Bit Input Shift Register SPICE model response.

2. Eight-Bit Data Latch

The SPICE model for the eight-bit data latch implements the structural design of Figure 100. A full source code listing is provided in Figure 117. Input for checking the data latch is shown in Figure 118. Various command values are presented to the data latch and a clock pulse provides the latch command. The eight-bit data latch response in Figure 119 shows that the various values are locked onto the command bus as required. The periodic clearing of all bits is caused by a reset signal that is pulsed at every other latch cycle.

```
* latch8.cir ==>  Eight-Bit Data Latch Transient Characteristics

* Logic Gate model definitions
.INCLUDE subckt.cir

* Power Supplies
VDS 1 0 5

* Input Signals
Vi0  i0  0 PWL(0 5 299.5n 5 301.5n 0 499.5n 0 501.5n 5 1 5)
Vi1  i1  0 PWL(0 5 399.5n 5 401.5n 0 699.5n 0 701.5n 5 1 5)
Vi2  i2  0 PWL(0 5 499.5n 5 501.5n 0 899.5n 0 901.5n 5 1 5)
Vi3  i3  0 PWL(0 5 599.5n 5 601.5n 0 1099.5n 0 1101.5n 5 1 5)
Vi4  i4  0 PWL(0 5 699.5n 5 701.5n 0 1299.5n 0 1301.5n 5 1 5)
Vi5  i5  0 PWL(0 5 799.5n 5 801.5n 0 1499.5n 0 1501.5n 5 1 5)
Vi6  i6  0 PWL(0 5 899.5n 5 901.5n 0 1699.5n 0 1701.5n 5 1 5)
Vi7  i7  0 PWL(0 5 999.5n 5 1001.5n 0 1899.5n 0 1901.5n 5 1 5)
Vrst rst 0 PULSE(5 0 2.5n 1n 1n 95n 100n)
Vlat latch 0 PULSE(0 5 24.5n 1n 1n 24n 50n)

* Eight-bit data latch
Xd0 i0 latch nR q0 nq0 1 0 DFLOPGC
Xd1 i1 latch nR q1 nq1 1 0 DFLOPGC
Xd2 i2 latch nR q2 nq2 1 0 DFLOPGC
Xd3 i3 latch nR q3 nq3 1 0 DFLOPGC
Xd4 i4 latch nR q4 nq4 1 0 DFLOPGC
Xd5 i5 latch nR q5 nq5 1 0 DFLOPGC
Xd6 i6 latch nR q6 nq6 1 0 DFLOPGC
Xd7 i7 latch nR q7 nq7 1 0 DFLOPGC
Xi rst nR 1 0 INV

* Simulation Parameters
.TRAN .1n 2000n 0 1n

.END
```

Figure 117. Eight-Bit Data Latch SPICE model source code.

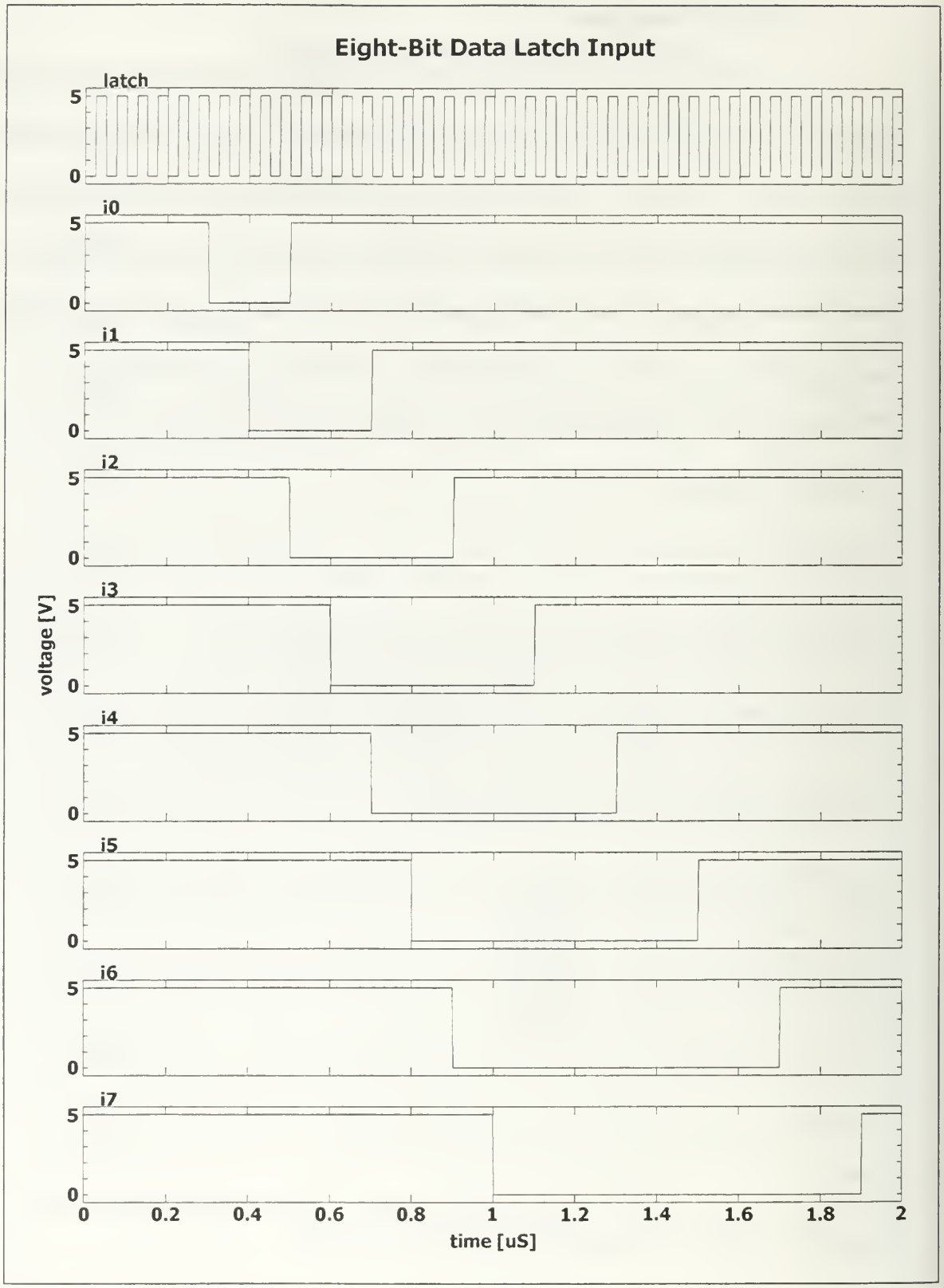


Figure 118. Eight-Bit Data Latch SPICE model input.

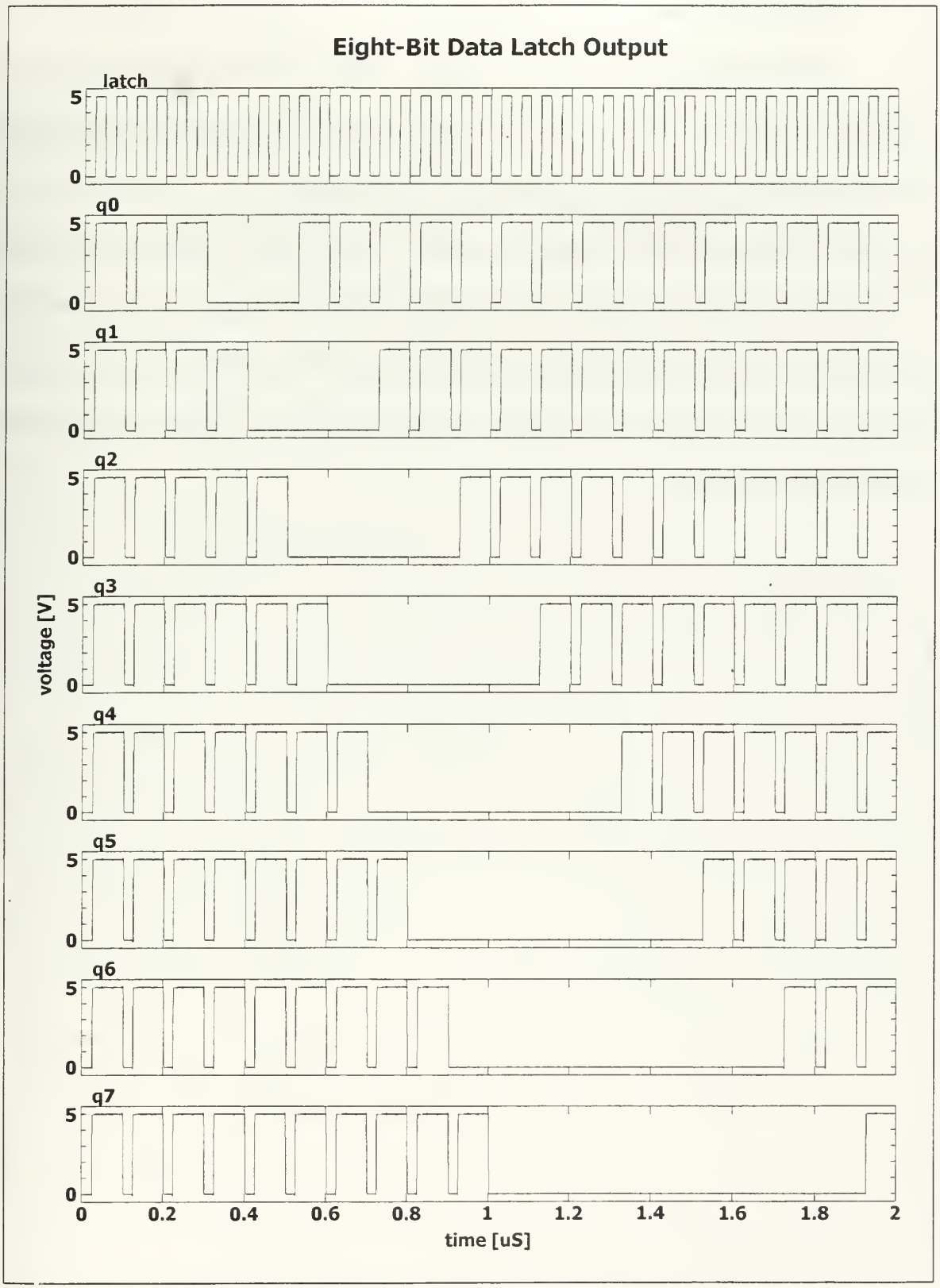


Figure 119. Eight-Bit Data Latch SPICE model response.

3. Input Stream Validity Check

The SPICE model for the input stream validity check implements the structural design of Figure 101. A full source code listing is provided in Figure 120. Input for testing the validity checker is shown in Figure 121. The command packet "0 0 1 1 1 1 1 1 1 0 1" is simulated shifting through the input shift register. The input stream validity check response in Figure 122 shows that a latch signal is generated when the command packet formatting requirements are met. This locks the command byte onto the command bus and presents a bus data valid signal for ten clock cycles. When the bus data valid flag is set, the partial clear signal is produced.

```
* invalid.cir ==>  Input Stream Validity Check Transient Characteristics

* Logic Gate model definitions
.INCLUDE subckt.cir

* Power Supplies
VDS 1 0 5

* Input Signals
Vi0  i0  0 PWL(0 0 25n 0 26n 5 325n 5 326n 0 425n 0 426n 5 775n 5 776n 0
               825n 0 826n 5 1 5)
Vi1  i1  0 PWL(0 0 75n 0 76n 5 375n 5 376n 0 475n 0 476n 5 825n 5 826n 0
               875n 0 876n 5 1 5)
Vi2  i2  0 PWL(0 0 125n 0 126n 5 425n 5 426n 0 525n 0 526n 5 860n 5 861n 0
               925n 0 926n 5 1 5)
Vi3  i3  0 PWL(0 0 175n 0 176n 5 475n 5 476n 0 575n 0 576n 5 860n 5 861n 0
               975n 0 976n 5 1 5)
Vi4  i4  0 PWL(0 0 225n 0 226n 5 525n 5 526n 0 625n 0 626n 5 860n 5 861n 0
               1025n 0 1026n 5 1 5)
Vi5  i5  0 PWL(0 0 275n 0 276n 5 575n 5 576n 0 675n 0 676n 5 860n 5 861n 0
               1075n 0 1076n 5 1 5)
Vi6  i6  0 PWL(0 0 325n 0 326n 5 625n 5 626n 0 725n 0 726n 5 860n 5 861n 0
               1125n 0 1126n 5 1 5)
Vi7  i7  0 PWL(0 0 375n 0 376n 5 675n 5 676n 0 775n 0 776n 5 860n 5 861n 0
               1175n 0 1176n 5 1 5)
Vi8  i8  0 PWL(0 0 425n 0 426n 5 725n 5 726n 0 825n 0 826n 5 860n 5 861n 0
               1225n 0 1226n 5 1 5)
Vi9  i9  0 PWL(0 0 475n 0 476n 5 775n 5 776n 0 1275n 0 1276n 5 1 5)
Vi10 i10 0 PWL(0 0 525n 0 526n 5 825n 5 826n 0 1325n 0 1326n 5 1 5)
Vi11 i11 0 PWL(0 0 575n 0 576n 5 860n 5 861n 0 1375n 0 1376n 5 1 5)
Vrst rst 0 PWL(0 0 1795n 0 1796n 5 1799n 5 1800n 0)
Vclk clk 0 PULSE(0 5 24.5n 1n 1n 24n 50n)

* Input stream validity check
Xx0 i1 i2 xo0 1 0 XOR2
Xx1 i3 i4 xo1 1 0 XOR2
Xx2 i5 i6 xo2 1 0 XOR2
Xx3 i7 i8 xo3 1 0 XOR2
Xx4 i9 xo3 xo4 1 0 XOR2
Xx5 xo0 xo1 xo5 1 0 XOR2
Xx6 xo2 xo4 xo6 1 0 XOR2
Xx7 xo5 xo6 xo7 1 0 XOR2
Xa0 i0 i11 ao0 1 0 NAND2
Xa1 ni10 xo7 ao1 1 0 NAND2
Xn0 ao0 ao1 frm 1 0 NOR2
Xi0 i10 ni10 1 0 INV
Xi1 frm nfrm 1 0 INV
Xn1 nbdv ni10 i11 no1 1 0 NOR3
Xn2 nfrm clk latch 1 0 NOR2
Xn3 rst no1 no3 1 0 NOR2
Xd0 frm latch no3 bdv nbdv 1 0 DFLOPGC
Xa2 bdv frm pC 1 0 AND2

* Simulation Parameters
.TRAN .1n 2000n 0 1n

.END
```

Figure 120. Input Stream Validity Check SPICE model source code.

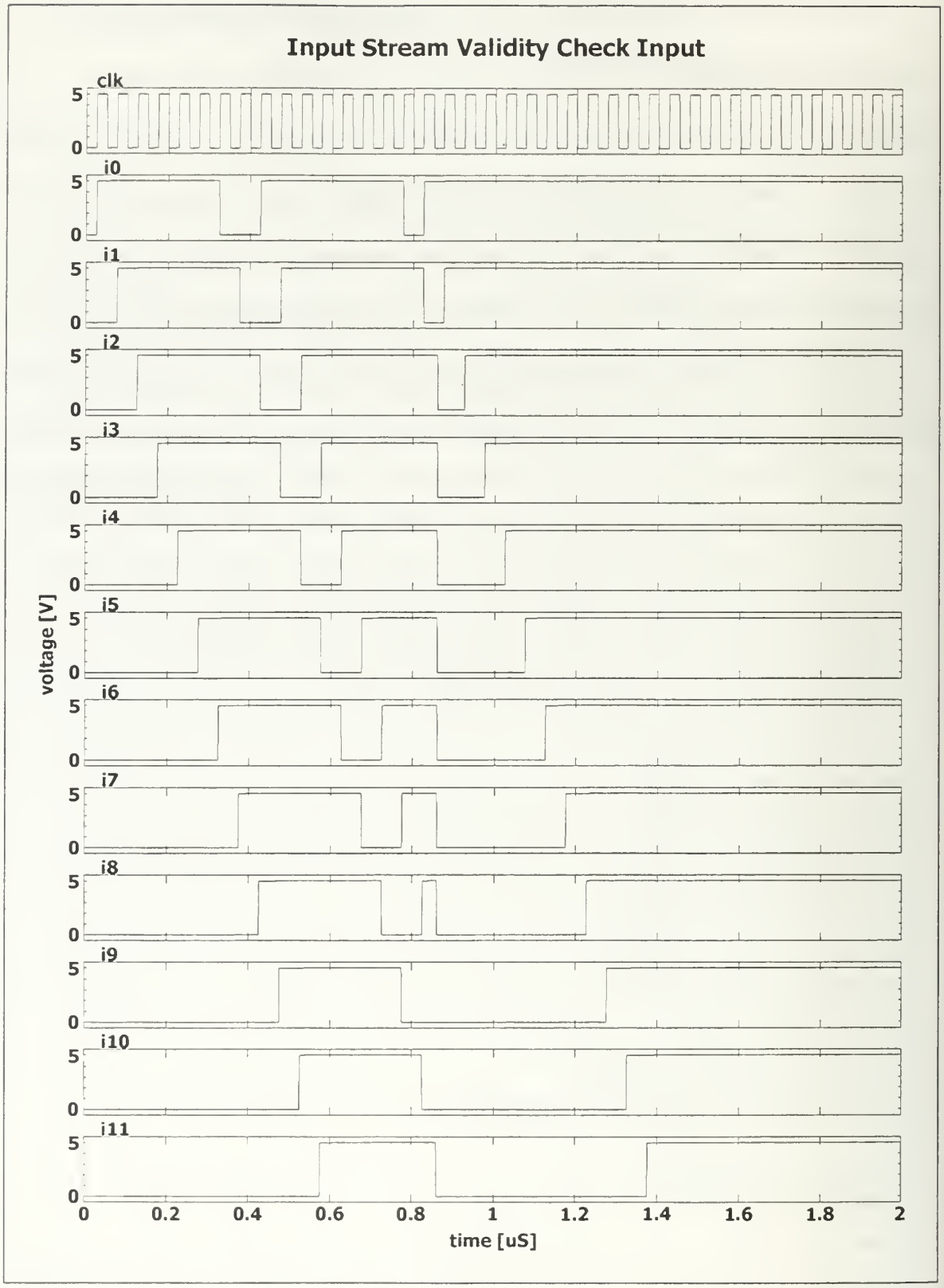


Figure 121. Input Stream Validity Check SPICE model input.

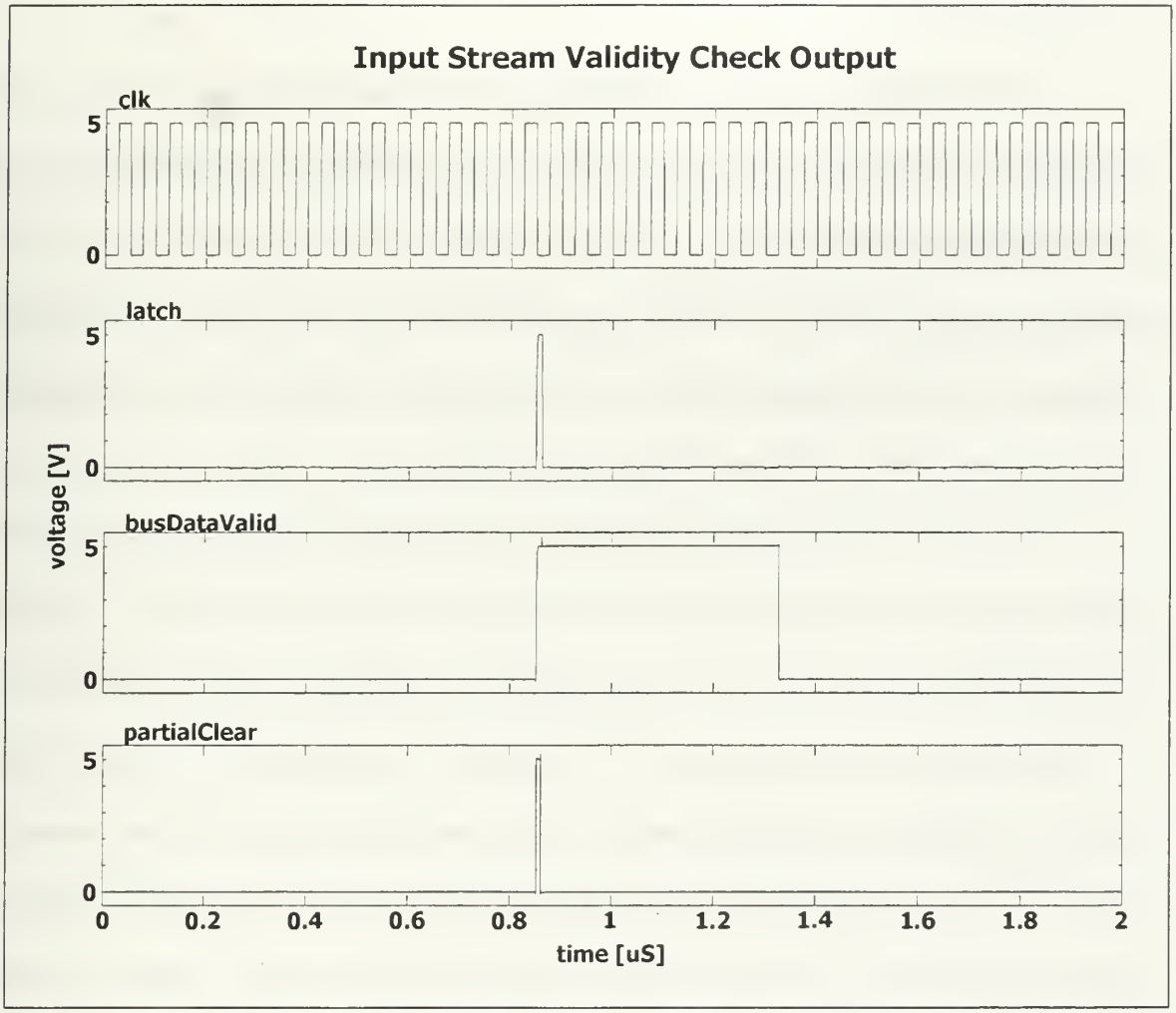


Figure 122. Input Stream Validity Check SPICE model response.

C. COMMAND DECODER AND CONTROLLER

The SPICE model for the Command Decoder and Controller is a combination of the models for its subordinate components. A full source code listing is provided in Figure 123. The command sequence 0 1 0 0 1 1 0 0 - 1 0 0 0 1 1 0 0 - 1 1 0 0 1 1 0 0 - 0 1 1 1 1 1 1 1 is simulated on the command bus with the corresponding bus valid flag. The Command Decoder and Controller control signal response is shown in Figure 124 and the register value response is shown in Figure 125.

The control signal response in Figure 124 illustrates the operating state transitions and the command signals generated by the Command Decoder and Controller. The initial command byte corresponds to the assigned address and causes the controller to shift to state 1. When the second byte is received, it is decoded as a command to set the pulse width value, which shifts the controller to state 3. A pulse width latch signal is issued because the pulse width register does not match the command bus value. Once the value is locked into the register, the falling pulse width difference flag clears the latch signal. Output is enabled when the new pulse width is latched, allowing the Tactor Power Control module to start using the new pulse width value. The third byte decodes as a command to set the repetition period, which requires no change in the control state. A repetition period latch command is issued because the repetition period register does not match the command bus value. Once the value is locked into the register, the falling repetition period difference flag clears the latch signal. The fourth byte is the "all call" address. The previous set of register commands is now complete and the controller shifts to state 0 to process the next command set because the command received in control state 3 is an address command. The next clock cycle shifts the controller to state 1 because the "all call" is a valid address and the controller

is now in state 0. The controller shift to state 0 just before 2 μS is caused by a system reset signal, which also disables the output flag and clears the storage registers.

The register response in Figure 125 illustrates how the stored values change in the pulse width and repetition period registers. At 0.5 μS, the pulse width latch signal caused the pulse width register to lock in the commanded value. At 1 μS, the repetition period latch signal caused the repetition period register to lock in the commanded value. Just before 2 μS, a system reset signal clears both storage registers.

```
* CmdDecodeCont.cir ==>  Command Decoder & Controller Transient Character.

* Logic Gate model definitions
.INCLUDE subckt.cir

* Power Supplies
VDS 1 0 5

* Input Signals
Vi0  i0  0 PWL(0 0 1450n 0 1451n 5 1 5)
Vi1  i1  0 PWL(0 0 1450n 0 1451n 5 1 5)
Vi2  i2  0 PWL(0 5 1 5)
Vi3  i3  0 PWL(0 5 1 5)
Vi4  i4  0 PWL(0 0 1450n 0 1451n 5 1 5)
Vi5  i5  0 PWL(0 0 1450n 0 1451n 5 1 5)
Vi6  i6  0 PWL(0 5 450n 5 451n 0 950n 0 951n 5 1 5)
Vi7  i7  0 PWL(0 0 450n 0 451n 5 1450n 5 1451n 0 1 0)
VbV bV 0 PWL(0 0 14n 0 15n 5 400n 5 401n 0 500n 0 501n 5 900n 5 901n 0 1000n
0 1001n 5 1400n 5 1401n 0 1500n 0 1501n 5 1900n 5 1901n 0 1 0)
Vrst nRst 0 PWL(0 0 10n 0 11n 5 1975n 5 1976n 0 1979n 0 1980n 5)
Vclk clk 0 PULSE(0 5 24.5n 1n 1n 24n 50n)
Va0 ad0 0 0
Va1 ad1 0 0
Va2 ad2 0 5
Va3 ad3 0 5
Va4 ad4 0 0
Va5 ad5 0 0
Va6 ad6 0 5

* Command sequence controller
Xcd0 aco2 clk nRst s0 ns0 1 0 DFLOPGC
Xcd1 ico0 clk nRst s1 ns1 1 0 DFLOPGC
Xca0 i7 s0 aco0 1 0 NAND2
Xca1 ns1 s0 aco1 1 0 NAND2
Xca2 aco0 aco1 oco0 aco2 1 0 NAND3
Xci0 aco0 ico0 1 0 INV
Xco0 i7 s0 nValidAddr oco0 1 0 OR3
Xca3 i7 bV s0 s1 aco3 1 0 AND4
Xci1 i6 ni6 1 0 INV
Xca4 i7 bV ni6 aco4 1 0 AND3
Xca5 aco4 oco1 oco2 aco5 1 0 NAND3
Xca6 nRst aco5 aco6 1 0 NAND2
Xco1 i3 i4 i5 oco1 1 0 NOR3
Xco2 i0 i1 i2 oco2 1 0 NOR3
Xca7 ni6 pwDiff aco3 aco7 1 0 NAND3
Xca8 aco3 i6 rpDiff aco8 1 0 NAND3
Xca9 aco3 pwDiff rpDiff aco9 1 0 NAND3
Xca10 aco7 aco8 aco9 aco10 1 0 NAND3
Xco3 aco6 aco10 oco3 1 0 OR2
Xco4 oco3 oco5 enOut 1 0 NOR2
Xco5 aco3 enOut oco5 1 0 NOR2
Xci2 aco7 pwLat 1 0 INV
Xci3 aco8 rpLat 1 0 INV
```

Figure 123. Command Decoder and Controller SPICE model source code.

```
* Address comparator
Xax0 i0 ad0 xao0 1 0 XNOR2
Xax1 i1 ad1 xao1 1 0 XNOR2
Xax2 i2 ad2 xao2 1 0 XNOR2
Xax3 i3 ad3 xao3 1 0 XNOR2
Xax4 i4 ad4 xao4 1 0 XNOR2
Xax5 i5 ad5 xao5 1 0 XNOR2
Xax6 i6 ad6 xao6 1 0 XNOR2
Xaa0 xao0 xao1 xao2 aao0 1 0 NAND3
Xaa1 xao3 xao4 xao5 aao1 1 0 NAND3
Xaa2 xao6 ni7 aao2 1 0 NAND2
Xaa3 i6 ni7 aao3 1 0 NAND2
Xaa4 i3 i4 i5 aao4 1 0 NAND3
Xaa5 i0 i1 i2 aao5 1 0 NAND3
Xao0 aao0 aao1 aao2 oao0 1 0 NOR3
Xao1 aao3 aao4 aao5 oao1 1 0 NOR3
Xao2 oao0 oao1 nValidAddr 1 0 NOR2
Xai i7 ni7 1 0 INV

* Pulse width register
Xpd0 i0 pwLat nRst pw0 npw0 1 0 DFLOPGC
Xpd1 i1 pwLat nRst pw1 npw1 1 0 DFLOPGC
Xpd2 i2 pwLat nRst pw2 npw2 1 0 DFLOPGC
Xpd3 i3 pwLat nRst pw3 npw3 1 0 DFLOPGC
Xpd4 i4 pwLat nRst pw4 npw4 1 0 DFLOPGC
Xpd5 i5 pwLat nRst pw5 npw5 1 0 DFLOPGC
Xpx0 i0 pw0 xpo0 1 0 XOR2
Xpx1 i1 pw1 xpo1 1 0 XOR2
Xpx2 i2 pw2 xpo2 1 0 XOR2
Xpx3 i3 pw3 xpo3 1 0 XOR2
Xpx4 i4 pw4 xpo4 1 0 XOR2
Xpx5 i5 pw5 xpo5 1 0 XOR2
Xpo0 xpo0 xpo1 xpo2 opo0 1 0 NOR3
Xpo1 xpo3 xpo4 xpo5 opo1 1 0 NOR3
Xpa0 opo0 opo1 pwDiff 1 0 NAND2

* Repetition period register
Xrd0 i0 rpLat nRst rp0 nrp0 1 0 DFLOPGC
Xrd1 i1 rpLat nRst rp1 nrp1 1 0 DFLOPGC
Xrd2 i2 rpLat nRst rp2 nrp2 1 0 DFLOPGC
Xrd3 i3 rpLat nRst rp3 nrp3 1 0 DFLOPGC
Xrd4 i4 rpLat nRst rp4 nrp4 1 0 DFLOPGC
Xrd5 i5 rpLat nRst rp5 nrp5 1 0 DFLOPGC
Xrx0 i0 rp0 xro0 1 0 XOR2
Xrx1 i1 rp1 xro1 1 0 XOR2
Xrx2 i2 rp2 xro2 1 0 XOR2
Xrx3 i3 rp3 xro3 1 0 XOR2
Xrx4 i4 rp4 xro4 1 0 XOR2
Xrx5 i5 rp5 xro5 1 0 XOR2
Xro0 xro0 xro1 xro2 oro0 1 0 NOR3
Xro1 xro3 xro4 xro5 oro1 1 0 NOR3
Xra0 oro0 oro1 rpDiff 1 0 NAND2

* Simulation Parameters
.TRAN .1n 2000n 0 1n

.END
```

Figure 123. Command Decoder and Controller SPICE model source code. (continued)

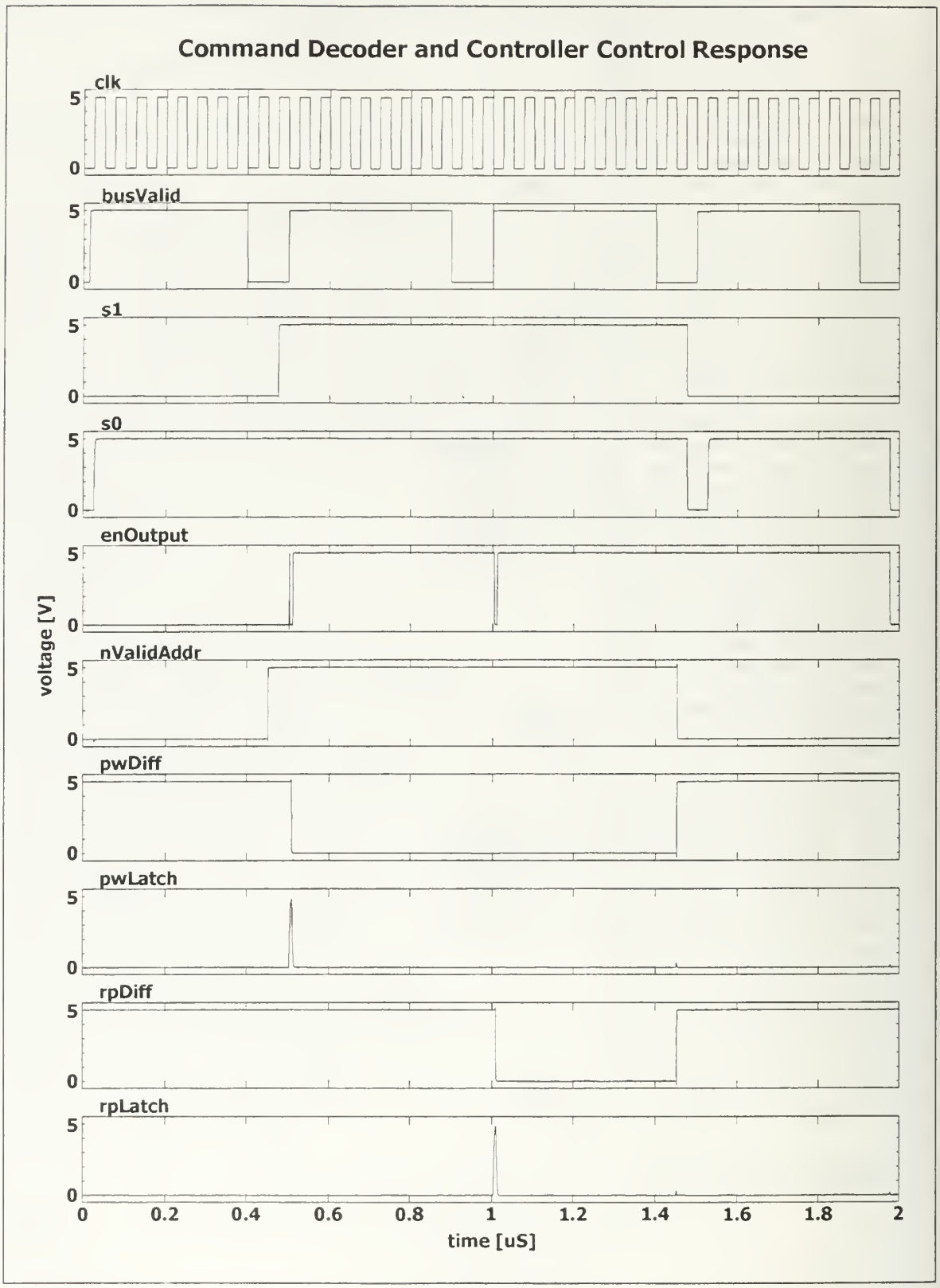


Figure 124. Command Decoder and Controller SPICE model Control response.

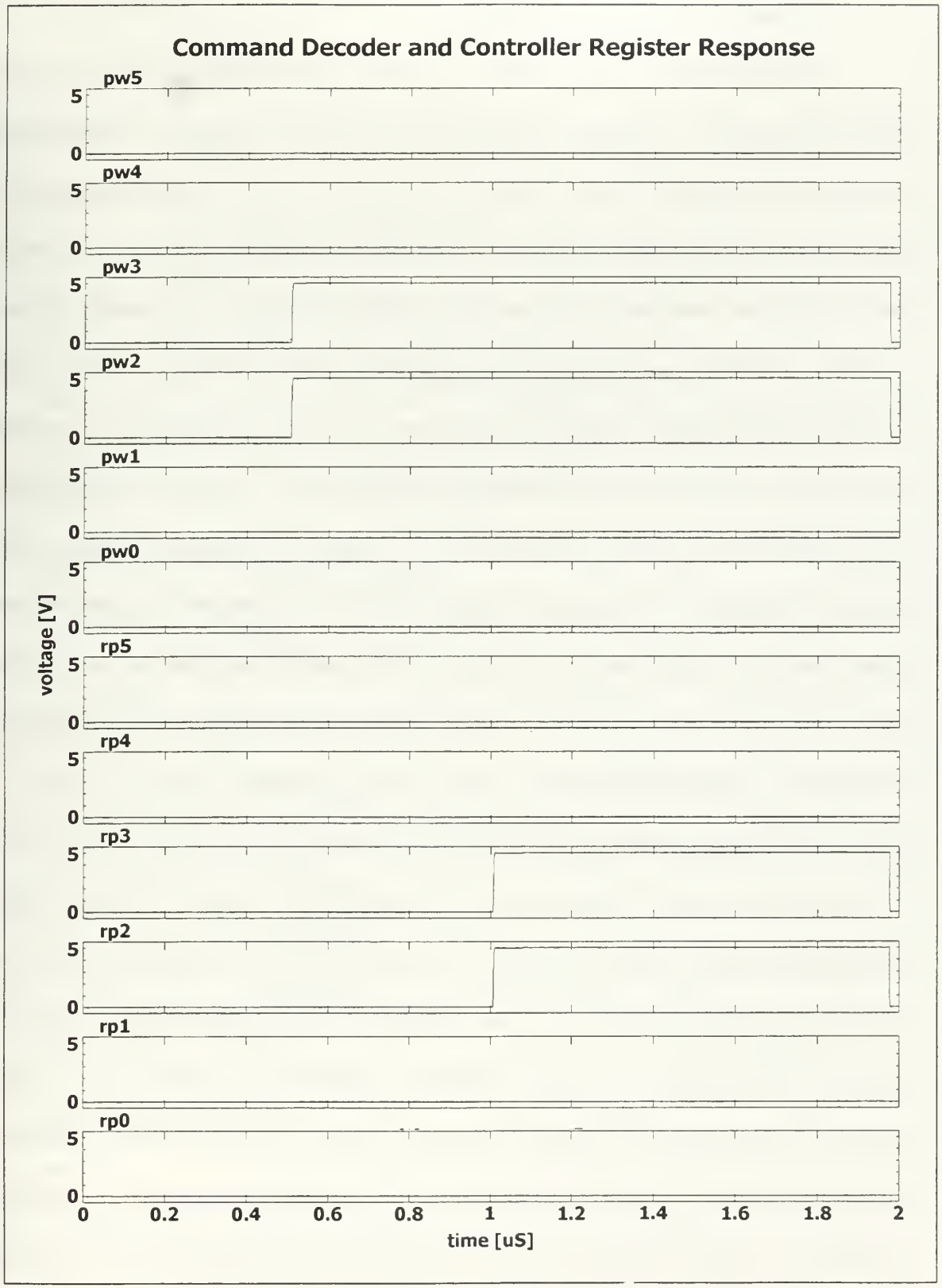


Figure 125. Command Decoder and Controller SPICE model Register response.

1. Command Sequence Controller

The SPICE model for the command sequence controller implements the structural design of Figure 102. A full source code listing is provided in Figure 126. The command sequence 0 1 0 0 1 1 0 0 - 1 0 0 0 1 1 0 0 - 1 1 0 0 1 1 0 0 - 0 1 1 1 1 1 1 1 is simulated on the command bus with the corresponding bus valid flag. The response in Figure 127 illustrates the operating state transitions and the command signals generated by the command sequence controller. The initial command byte corresponds to the assigned address and causes the controller to shift to state 1. When the second byte is received, it is decoded as a command to set the pulse width value, which shifts the controller to state 3. A pulse width latch signal is now issued since the pulse width register does not match the command bus value. Once the value is locked into the register, the falling pulse width difference flag clears the latch signal. Output is enabled when the new pulse width is latched, allowing the Tactor Power Control module to start using the new pulse width value. The third byte decodes as a command to set the repetition period, which requires no change in the control state. A repetition period latch command is issued since the repetition period register does not match the command bus value,. Once the value is locked into the register, the falling repetition period difference flag clears the latch signal. The fourth byte is the "all call" address. The previous set of register commands is now complete and the controller shifts to state 0 to process the next command set because an address command was received in control state 3. The next clock cycle shifts the controller to state 1 since the "all call" is a valid address and the controller is now in state 0. The controller shift to state 0 just before 2 μS in Figure 127 is caused by a system reset signal, which also disables the output flag and clears the storage registers.

```
* CmdSeqCont.cir ==>  Command Sequence Controller Transient Characteristics

* Logic Gate model definitions
.INCLUDE subckt.cir

* Power Supplies
VDS 1 0 5

* Input Signals
Vi0  i0  0 PWL(0 0 1450n 0 1451n 5 1 5)
Vi1  i1  0 PWL(0 0 1450n 0 1451n 5 1 5)
Vi2  i2  0 PWL(0 5 1 5)
Vi3  i3  0 PWL(0 5 1 5)
Vi4  i4  0 PWL(0 0 1450n 0 1451n 5 1 5)
Vi5  i5  0 PWL(0 0 1450n 0 1451n 5 1 5)
Vi6  i6  0 PWL(0 5 450n 5 451n 0 950n 0 951n 5 1 5)
Vi7  i7  0 PWL(0 0 450n 0 451n 5 1450n 5 1451n 0 1 0)
VnVA nVA 0 PWL(0 0 450n 0 451n 5 1450n 5 1451n 0 1 0)
VbV bV 0 PWL(0 5 400n 5 401n 0 500n 0 501n 5 900n 5 901n 0 1000n 0 1001n 5
             1400n 5 1401n 0 1500n 0 1501n 5 1900n 5 1901n 0 1 0)
VpwD pwD 0 PWL(0 5 600n 5 601n 0 951n 0 952n 5 1 5)
VrpD rpD 0 PWL(0 5 1100n 5 1101n 0 1451n 0 1452n 5 1 5)
Vrst nRst 0 PWL(0 5 1975n 5 1976n 0 1979n 0 1980n 5)
Vclk clk 0 PULSE(0 5 24.5n 1n 1n 24n 50n)

* Command sequence controller
Xd0 ao2 clk nRst q0 nq0 1 0 DFLOPGC
Xd1 io0 clk nRst q1 nq1 1 0 DFLOPGC
Xa0 i7 q0 ao0 1 0 NAND2
Xa1 nq1 q0 ao1 1 0 NAND2
Xa2 ao0 ao1 oo0 ao2 1 0 NAND3
Xi0 ao0 io0 1 0 INV
Xo0 i7 q0 nVA oo0 1 0 OR3
Xa3 i7 bV q0 q1 ao3 1 0 AND4
Xi1 i6 ni6 1 0 INV
Xa4 i7 bV ni6 ao4 1 0 AND3
Xa5 ao4 oo1 oo2 ao5 1 0 NAND3
Xa6 nRst ao5 ao6 1 0 NAND2
Xo1 i3 i4 i5 oo1 1 0 NOR3
Xo2 i0 i1 i2 oo2 1 0 NOR3
Xa7 ni6 pwD ao3 ao7 1 0 NAND3
Xa8 ao3 i6 rpD ao8 1 0 NAND3
Xa9 ao3 pwD rpD ao9 1 0 NAND3
Xa10 ao7 ao8 ao9 ao10 1 0 NAND3
Xo3 ao6 ao10 oo3 1 0 OR2
Xo4 oo3 oo5 enOut 1 0 NOR2
Xo5 ao3 enOut oo5 1 0 NOR2
Xi2 ao7 pwLat 1 0 INV
Xi3 ao8 rpLat 1 0 INV

* Simulation Parameters
.TRAN .1n 2000n 0 1n

.END
```

Figure 126. Command Sequence Controller SPICE model source code.

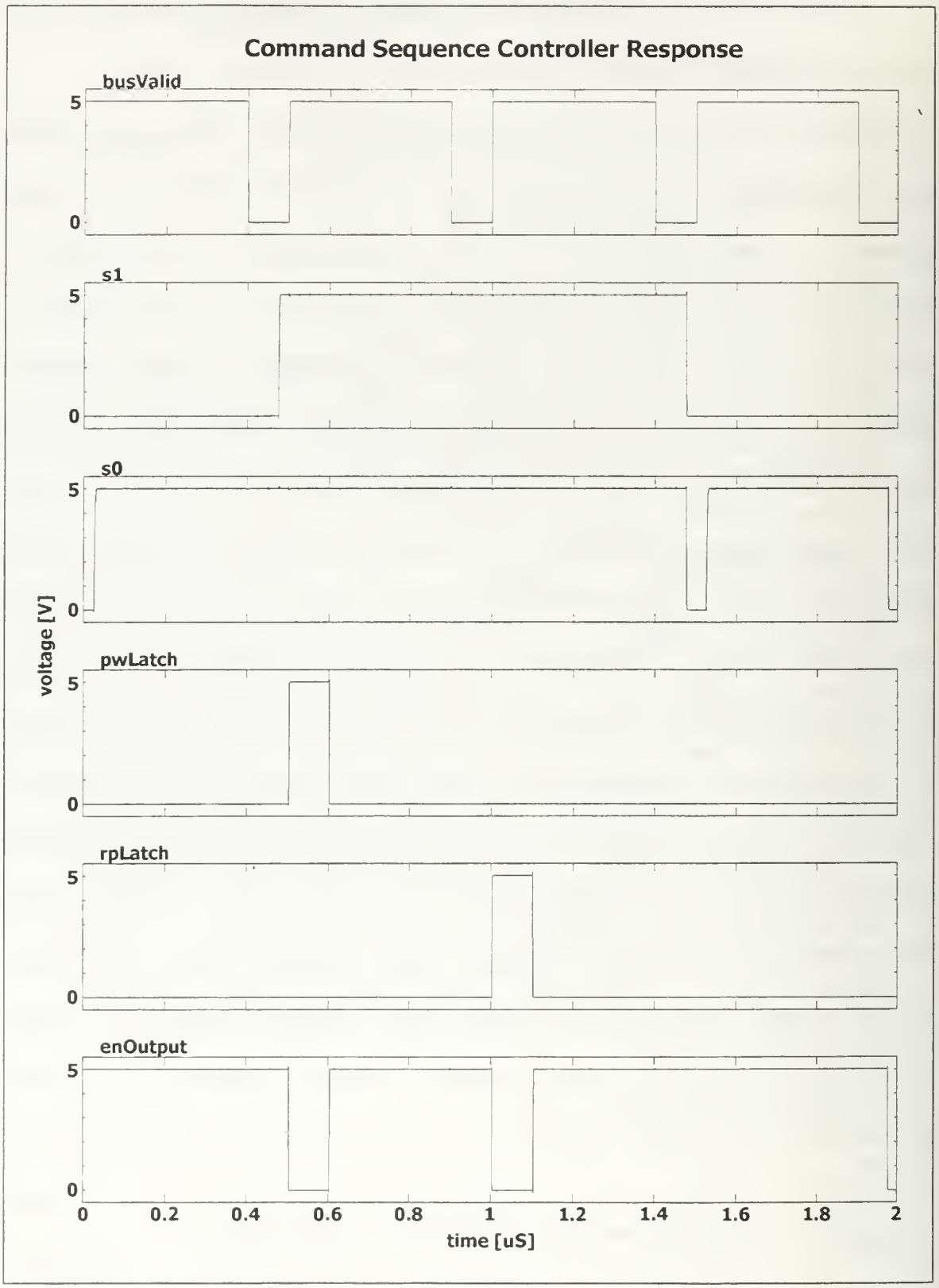


Figure 127. Command Sequence Controller SPICE model response.

2. Address Comparator

The SPICE model for the address comparator implements the structural design of Figure 103. A full source code listing is provided in Figure 128. The reference address is set to " 1 0 0 1 1 0 0" and various values are presented on the command bus. The address comparator response in Figure 129 illustrates the continuous address verification. The first and last valid address indications correspond to the all-call address and the middle two valid address indications are due to the assigned tactor address.

```
* AddrComp.cir ==>  Address Comparator Transient Characteristics

* Logic Gate model definitions
.INCLUDE subckt.cir

* Power Supplies
VDS 1 0 5

* Input Signals
Vi0  i0  0 PWL(0 5 299.5n 5 300.5n 0 1299.5n 0 1300.5n 5 1 5)
Vi1  i1  0 PWL(0 5 299.5n 5 300.5n 0 1399.5n 0 1400.5n 5 1 5)
Vi2  i2  0 PWL(0 5 299.5n 5 300.5n 0 699.5n 0 700.5n 5 1699.5n 5 1700.5n 0
               1 0)
Vi3  i3  0 PWL(0 5 299.5n 5 300.5n 0 599.5n 0 600.5n 5 1 5)
Vi4  i4  0 PWL(0 5 299.5n 5 300.5n 0 1399.5n 0 1400.5n 5 1 5)
Vi5  i5  0 PWL(0 5 299.5n 5 300.5n 0 1199.5n 0 1200.5n 5 1 5)
Vi6  i6  0 PWL(0 5 299.5n 5 300.5n 0 499.5n 0 500.5n 5 1099.5n 5 1100.5n 0
               1599.5n 0 1600.5n 5 1 5)
Vi7  i7  0 PWL(0 5 199.5n 5 200.5n 0 899.5n 0 900.5n 5 999.5n 5 1000.5n 0
               1 0)
Va0 ad0 0 0
Va1 ad1 0 0
Va2 ad2 0 5
Va3 ad3 0 5
Va4 ad4 0 0
Va5 ad5 0 0
Va6 ad6 0 5

* Address comparator
Xx0 i0 ad0 xo0 1 0 XNOR2
Xx1 i1 ad1 xo1 1 0 XNOR2
Xx2 i2 ad2 xo2 1 0 XNOR2
Xx3 i3 ad3 xo3 1 0 XNOR2
Xx4 i4 ad4 xo4 1 0 XNOR2
Xx5 i5 ad5 xo5 1 0 XNOR2
Xx6 i6 ad6 xo6 1 0 XNOR2
Xa0 xo0 xo1 xo2 ao0 1 0 NAND3
Xa1 xo3 xo4 xo5 ao1 1 0 NAND3
Xa2 xo6 ni7 ao2 1 0 NAND2
Xa3 i6 ni7 ao3 1 0 NAND2
Xa4 i3 i4 i5 ao4 1 0 NAND3
Xa5 i0 i1 i2 ao5 1 0 NAND3
Xo0 ao0 ao1 ao2 oo0 1 0 NOR3
Xo1 ao3 ao4 ao5 oo1 1 0 NOR3
Xo2 oo0 oo1 nValidAddr 1 0 NOR2
Xi i7 ni7 1 0 INV

* Simulation Parameters
.TRAN .1n 2000n 0 1n

.END
```

Figure 128. Address Comparator SPICE model source code.

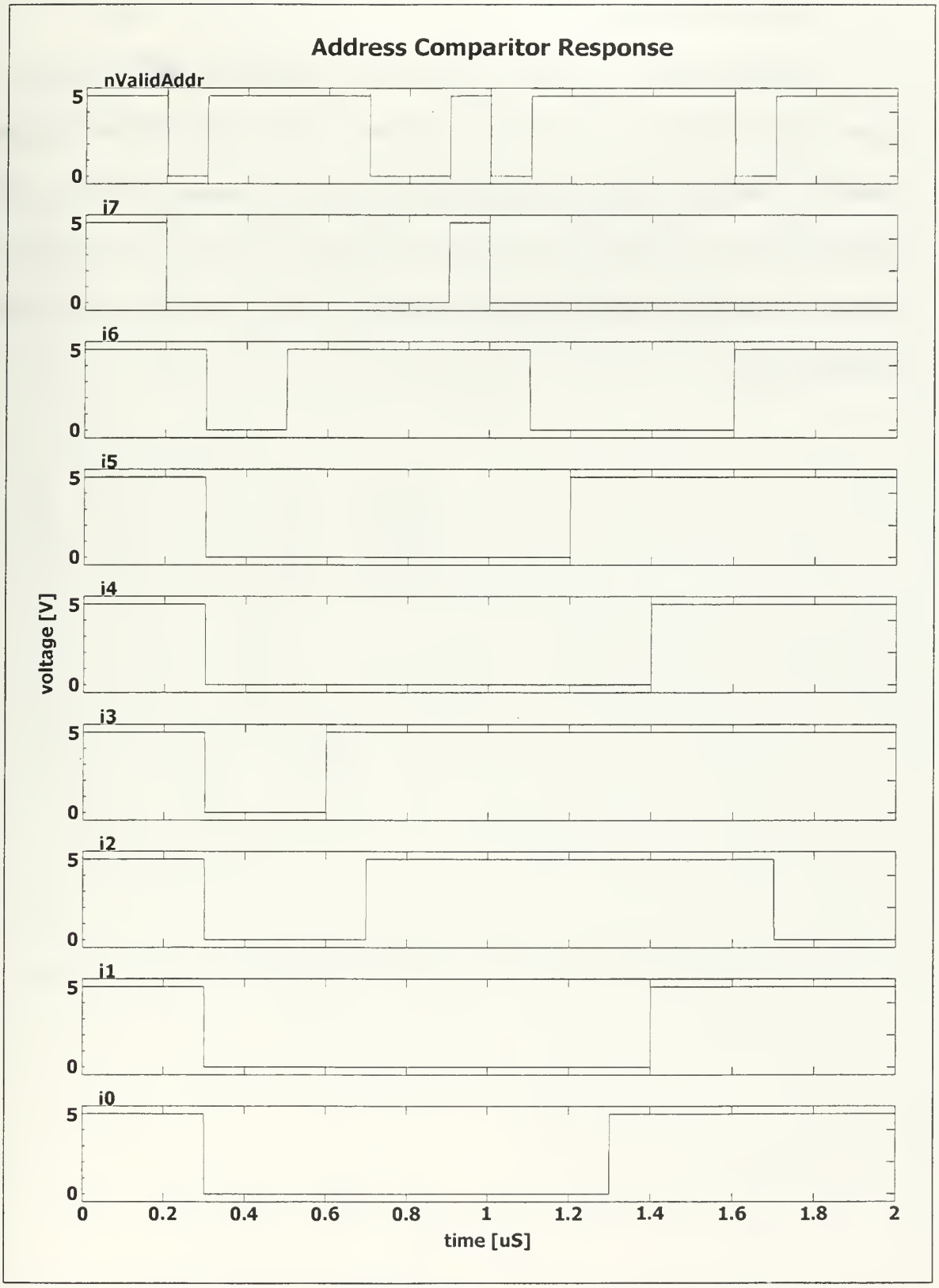


Figure 129. Address Comparator SPICE model response.

3. Pulse Width Register

The SPICE model for the pulse width register implements the structural design of Figure 104. A full source code listing is provided in Figure 130. Various values are presented to the register and a clock pulse provides the latch command. The pulse width register response in Figure 131 shows the various values are stored in the register as required. The periodic clearing of all bits is caused by a reset signal that is pulsed at every other latch cycle.

```
* PWregister.cir ==>  Pulse Width Register Transient Characteristics

* Logic Gate model definitions
.INCLUDE subckt.cir

* Power Supplies
VDS 1 0 5

* Input Signals
Vi0  i0  0 PWL(0 5 299.5n 5 301.5n 0 499.5n 0 501.5n 5 1 5)
Vi1  i1  0 PWL(0 5 399.5n 5 401.5n 0 699.5n 0 701.5n 5 1 5)
Vi2  i2  0 PWL(0 5 499.5n 5 501.5n 0 899.5n 0 901.5n 5 1 5)
Vi3  i3  0 PWL(0 5 599.5n 5 601.5n 0 1099.5n 0 1101.5n 5 1 5)
Vi4  i4  0 PWL(0 5 699.5n 5 701.5n 0 1299.5n 0 1301.5n 5 1 5)
Vi5  i5  0 PWL(0 5 799.5n 5 801.5n 0 1499.5n 0 1501.5n 5 1 5)
Vi6  i6  0 PWL(0 5 899.5n 5 901.5n 0 1699.5n 0 1701.5n 5 1 5)
Vi7  i7  0 PWL(0 5 999.5n 5 1001.5n 0 1899.5n 0 1901.5n 5 1 5)
Vrst rst 0 PULSE(5 0 2.5n 1n 1n 95n 100n)
Vclk latch 0 PULSE(0 5 24.5n 1n 1n 24n 50n)

* Pulse width register
Xd0 i0 latch nR pw0 npw0 1 0 DFLOPGC
Xd1 i1 latch nR pw1 npw1 1 0 DFLOPGC
Xd2 i2 latch nR pw2 npw2 1 0 DFLOPGC
Xd3 i3 latch nR pw3 npw3 1 0 DFLOPGC
Xd4 i4 latch nR pw4 npw4 1 0 DFLOPGC
Xd5 i5 latch nR pw5 npw5 1 0 DFLOPGC
Xx0 i0 pw0 xo0 1 0 XOR2
Xx1 i1 pw1 xo1 1 0 XOR2
Xx2 i2 pw2 xo2 1 0 XOR2
Xx3 i3 pw3 xo3 1 0 XOR2
Xx4 i4 pw4 xo4 1 0 XOR2
Xx5 i5 pw5 xo5 1 0 XOR2
Xo0 xo0 xo1 xo2 oo0 1 0 NOR3
Xo1 xo3 xo4 xo5 oo1 1 0 NOR3
Xa0 oo0 oo1 pwDiff 1 0 NAND2
Xi rst nR 1 0 INV

* Simulation Parameters
.TRAN .1n 2000n 0 1n

.END
```

Figure 130. Pulse Width Register SPICE model source code.

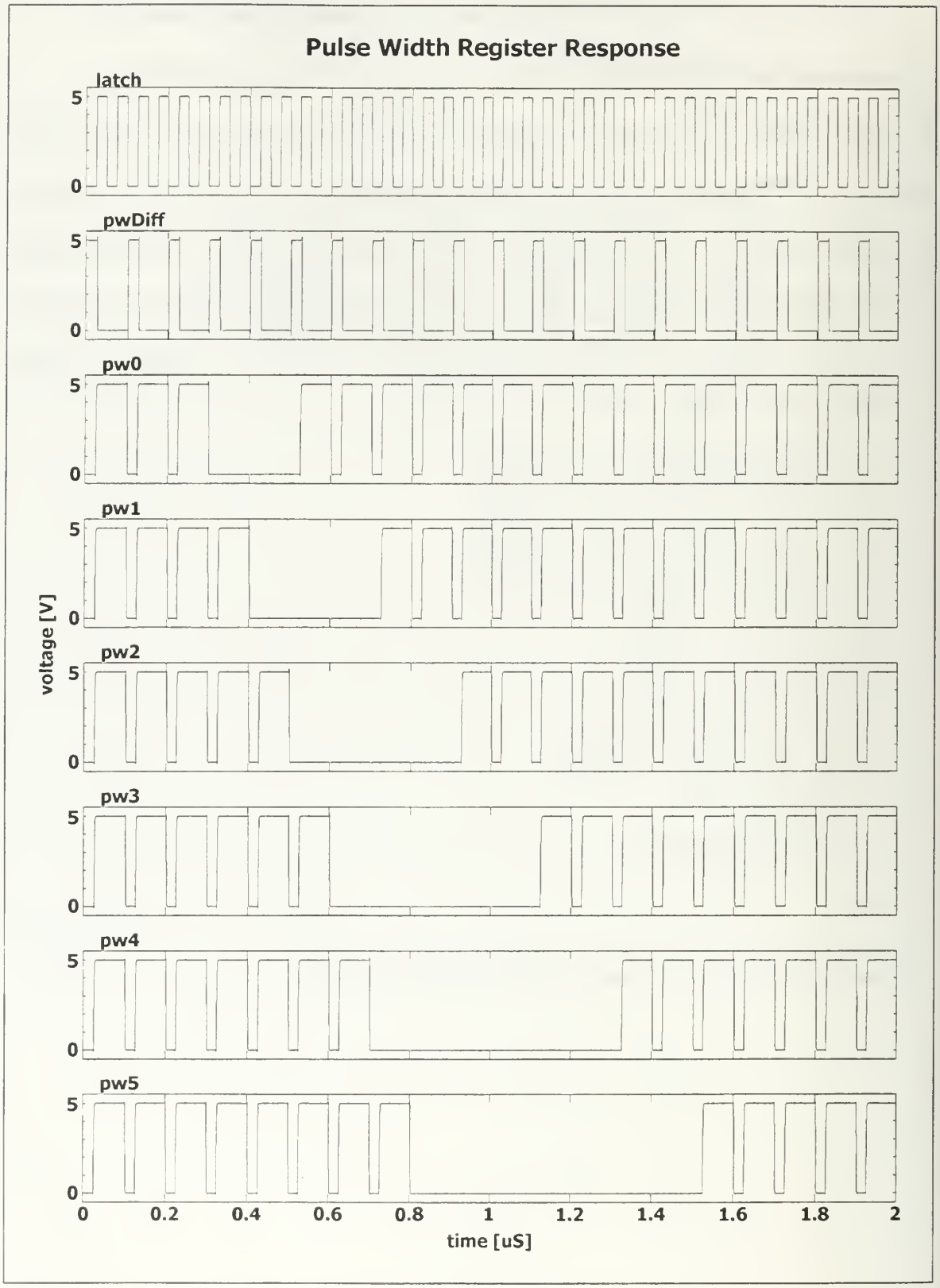


Figure 131. Pulse Width Register SPICE model response.

4. Repetition Period Register

The SPICE model for the repetition period register implements the structural design of Figure 105. A full source code listing is provided in Figure 132. Various values are presented to the register and a clock pulse provides the latch command. The repetition period register response in Figure 133 shows that the various values are stored in the register as required. The periodic clearing of all bits is caused by a reset signal that is pulsed at every other latch cycle.

```
* RPregister.cir ==>  Repetition Period Register Transient Characteristics

* Logic Gate model definitions
.INCLUDE subckt.cir

* Power Supplies
VDS 1 0 5

* Input Signals
Vi0  i0  0 PWL(0 5 299.5n 5 301.5n 0 499.5n 0 501.5n 5 1 5)
Vi1  i1  0 PWL(0 5 399.5n 5 401.5n 0 699.5n 0 701.5n 5 1 5)
Vi2  i2  0 PWL(0 5 499.5n 5 501.5n 0 899.5n 0 901.5n 5 1 5)
Vi3  i3  0 PWL(0 5 599.5n 5 601.5n 0 1099.5n 0 1101.5n 5 1 5)
Vi4  i4  0 PWL(0 5 699.5n 5 701.5n 0 1299.5n 0 1301.5n 5 1 5)
Vi5  i5  0 PWL(0 5 799.5n 5 801.5n 0 1499.5n 0 1501.5n 5 1 5)
Vi6  i6  0 PWL(0 5 899.5n 5 901.5n 0 1699.5n 0 1701.5n 5 1 5)
Vi7  i7  0 PWL(0 5 999.5n 5 1001.5n 0 1899.5n 0 1901.5n 5 1 5)
Vrst rst 0 PULSE(5 0 2.5n 1n 1n 95n 100n)
Vclk latch 0 PULSE(0 5 24.5n 1n 1n 24n 50n)

* Repetition period register
Xd0 i0 latch nR rp0 nrp0 1 0 DFLOPGC
Xd1 i1 latch nR rp1 nrp1 1 0 DFLOPGC
Xd2 i2 latch nR rp2 nrp2 1 0 DFLOPGC
Xd3 i3 latch nR rp3 nrp3 1 0 DFLOPGC
Xd4 i4 latch nR rp4 nrp4 1 0 DFLOPGC
Xd5 i5 latch nR rp5 nrp5 1 0 DFLOPGC
Xx0 i0 rp0 xo0 1 0 XOR2
Xx1 i1 rp1 xo1 1 0 XOR2
Xx2 i2 rp2 xo2 1 0 XOR2
Xx3 i3 rp3 xo3 1 0 XOR2
Xx4 i4 rp4 xo4 1 0 XOR2
Xx5 i5 rp5 xo5 1 0 XOR2
Xo0 xo0 xo1 xo2 oo0 1 0 NOR3
Xo1 xo3 xo4 xo5 oo1 1 0 NOR3
Xa0 oo0 oo1 rpDiff 1 0 NAND2
Xi rst nR 1 0 INV

* Simulation Parameters
.TRAN .1n 2000n 0 1n

.END
```

Figure 132. Repetition Period Register SPICE model source code.

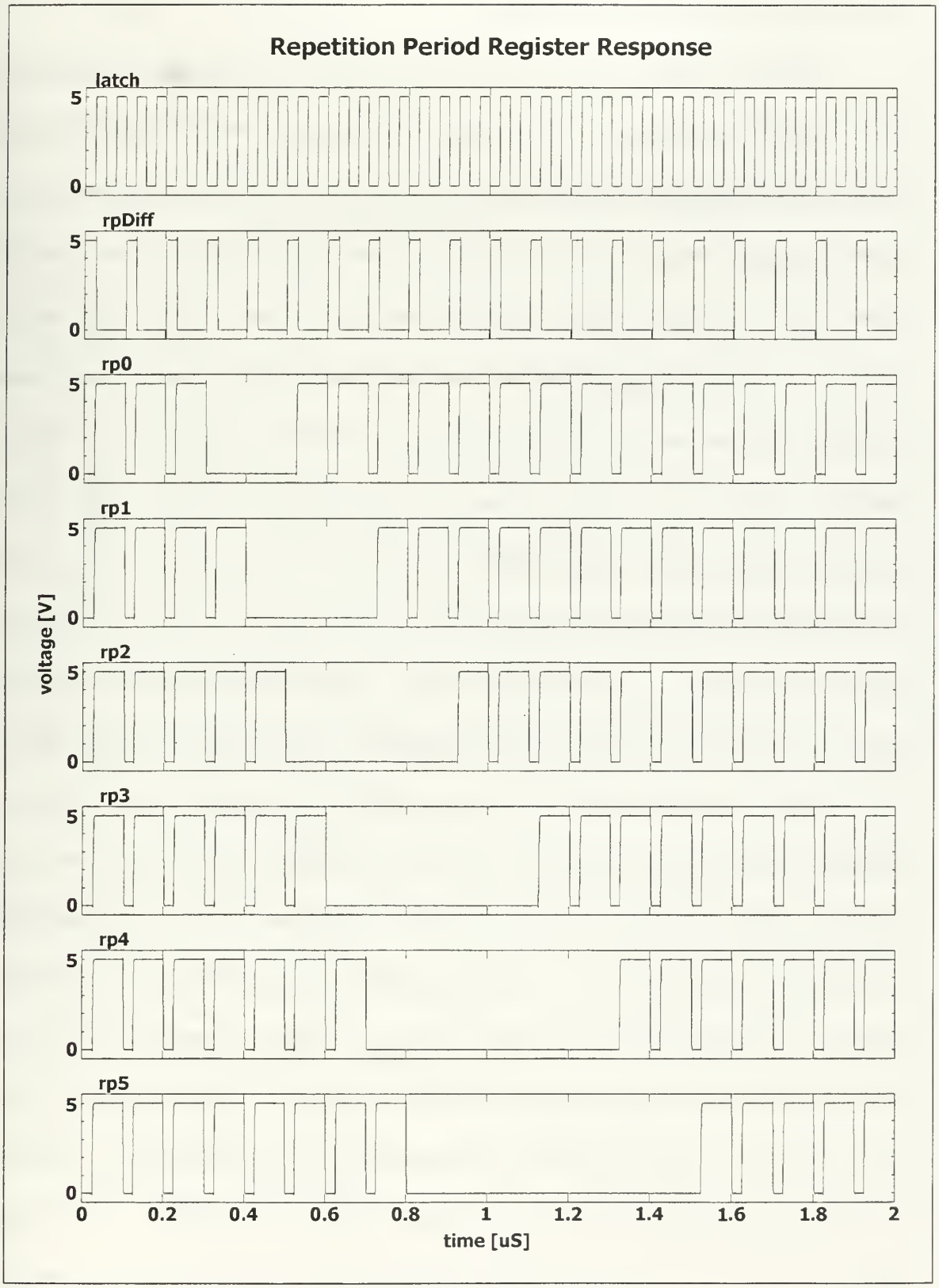


Figure 133. Repetition Period Register SPICE model response.

D. TACTOR POWER CONTROLLER

The SPICE model for the Tactor Power Controller is a combination of the models for its subordinate components. A full source code listing is provided in Figure 134. The input signals simulate receiving a register command setting the pulse width to 2 at 0.5 μS, a register command setting the repetition period to 1 at 1 μS, and finally a system reset just before 2 μS. The Tactor Power Controller response is shown in Figure 135 with an emphasis on the output signals and the pulse width and repetition period down counter operations. For the first 0.5 μS, the counters are continuously cleared since the output is disabled. From 0.5 to 1 μS, the counters are continuously loading the values in their respective storage registers. This condition is driven by the 0 value stored in the repetition period register and effectively causes the tactor to be continuously activated since the pulse width is now non-zero. At 1 μS, when the output is momentarily disabled, the pulse width counter is cleared and the output oscillation signals stop. When the new repetition period is set, the output is again enabled. The pulse width and repetition register values are immediately latched into the down counters and, since pulse width is non-zero, tactor activation begins. The down counters count down together and tactor activation stops when the pulse width count reaches zero. The pulse width down counter stops at zero while the repetition period down counter continues counting. When the repetition period count reaches one, the wave shape is complete and the counters are reloaded on the next clock cycle. This process continues, creating a 50% activation cycle for the simulated commands. Just before 2 μS, a system reset signal is simulated and tactor activation is immediately halted.

```
* TactorPwrCont.cir ==>  Tactor Power Controller Transient Characteristics

* Logic Gate model definitions
.INCLUDE subckt.cir

* Power Supplies
VDS 1 0 5

* Input Signals
VenOut enOut 0 PWL(0 0 505n 0 506n 5 1005n 5 1006n 0 1009n 0 1010n 5
                   1980n 5 1981n 0 1 0)
Vclk clk 0 PULSE(0 5 24.5n 1n 1n 49n 100n)
Vosc osc 0 PULSE(0 5 24.5n 1n 1n 24n 50n)
Vpw5 pw5  0 PWL(0 0 1 0)
Vpw4 pw4  0 PWL(0 0 1 0)
Vpw3 pw3  0 PWL(0 0 1 0)
Vpw2 pw2  0 PWL(0 0 1 0)
Vpw1 pw1  0 PWL(0 0 500n 0 501n 5 1980n 5 1981n 0 1 0)
Vpw0 pw0  0 PWL(0 0 1 0)
Vrp5 rp5  0 PWL(0 0 1 0)
Vrp4 rp4  0 PWL(0 0 1 0)
Vrp3 rp3  0 PWL(0 0 1 0)
Vrp2 rp2  0 PWL(0 0 1 0)
Vrp1 rp1  0 PWL(0 0 1 0)
Vrp0 rp0  0 PWL(0 0 1000n 0 1001n 5 1980n 5 1981n 0 1 0)

* Power control logic
Xi0 enOut cntClr 1 0 INV

* Pulse Width Down Counter
Xpa0 opo0 opo1 npwZ 1 0 NAND2
Xpo0 pq3 pq4 pq5 opo0 1 0 NOR3
Xpo1 pq0 pq1 pq2 opo1 1 0 NOR3
Xpm0 npq0 pw0 nrpGT1 mpo0 1 0 MUX
Xpd0 mpo0 clk opo5 pq0 npq0 1 0 DFLOPGC
Xpx1 npq0 npq1 xpo1 1 0 XNOR2
Xpm1 xpo1 pw1 nrpGT1 mpo1 1 0 MUX
Xpd1 mpo1 clk opo5 pq1 npq1 1 0 DFLOPGC
Xpa1 npq0 npq1 apo1 1 0 NAND2
Xpx2 apo1 pq2 xpo2 1 0 XNOR2
Xpm2 xpo2 pw2 nrpGT1 mpo2 1 0 MUX
Xpd2 mpo2 clk opo5 pq2 npq2 1 0 DFLOPGC
Xpo2 apo1 pq2 opo2 1 0 NOR2
Xpx3 opo2 npq3 xpo3 1 0 XNOR2
Xpm3 xpo3 pw3 nrpGT1 mpo3 1 0 MUX
Xpd3 mpo3 clk opo5 pq3 npq3 1 0 DFLOPGC
Xpa2 opo2 npq3 apo2 1 0 NAND2
Xpx4 apo2 pq4 xpo4 1 0 XNOR2
Xpm4 xpo4 pw4 nrpGT1 mpo4 1 0 MUX
Xpd4 mpo4 clk opo5 pq4 npq4 1 0 DFLOPGC
Xpo3 apo2 pq4 opo3 1 0 NOR2
Xpx5 opo3 npq5 xpo5 1 0 XNOR2
Xpm5 xpo5 pw5 nrpGT1 mpo5 1 0 MUX
Xpd5 mpo5 clk opo5 pq5 npq5 1 0 DFLOPGC
Xpo4 nrpGT1 npwZ opo4 1 0 NOR2
Xpo5 opo4 cntClr opo5 1 0 NOR2
```

Figure 134. Tactor Power Controller SPICE model source code.

```
* Repetition Period Down Counter
Xra0 oro0 oro1 oro2 nrpGT1 1 0 AND3
Xro0 rq1 rq2 oro0 1 0 NOR2
Xro1 rq3 rq4 oro1 1 0 NOR2
Xro2 rq5 rq6 rq7 oro2 1 0 NOR3
Xrm0 nrq0 0 nrpGT1 mro0 1 0 MUX
Xrd0 mro0 clk oro7 rq0 nrq0 1 0 DFLOPGC
Xrx1 nrq0 nrq1 xro1 1 0 XNOR2
Xrm1 xro1 0 nrpGT1 mro1 1 0 MUX
Xrd1 mro1 clk oro7 rq1 nrq1 1 0 DFLOPGC
Xra2 nrq0 nrq1 aro2 1 0 NAND2
Xrx2 aro2 rq2 xro2 1 0 XNOR2
Xrm2 xro2 rp0 nrpGT1 mro2 1 0 MUX
Xrd2 mro2 clk oro7 rq2 nrq2 1 0 DFLOPGC
Xro3 aro2 rq2 oro3 1 0 NOR2
Xrx3 oro3 nrq3 xro3 1 0 XNOR2
Xrm3 xro3 rp1 nrpGT1 mro3 1 0 MUX
Xrd3 mro3 clk oro7 rq3 nrq3 1 0 DFLOPGC
Xra3 oro3 nrq3 aro3 1 0 NAND2
Xrx4 aro3 rq4 xro4 1 0 XNOR2
Xrm4 xro4 rp2 nrpGT1 mro4 1 0 MUX
Xrd4 mro4 clk oro7 rq4 nrq4 1 0 DFLOPGC
Xro4 aro3 rq4 oro4 1 0 NOR2
Xrx5 oro4 nrq5 xro5 1 0 XNOR2
Xrm5 xro5 rp3 nrpGT1 mro5 1 0 MUX
Xrd5 mro5 clk oro7 rq5 nrq5 1 0 DFLOPGC
Xra4 oro4 nrq5 aro4 1 0 NAND2
Xrx6 aro4 rq6 xro6 1 0 XNOR2
Xrm6 xro6 rp4 nrpGT1 mro6 1 0 MUX
Xrd6 mro6 clk oro7 rq6 nrq6 1 0 DFLOPGC
Xro5 aro4 rq6 oro5 1 0 NOR2
Xrx7 oro5 nrq7 xro7 1 0 XNOR2
Xrm7 xro7 rp5 nrpGT1 mro7 1 0 MUX
Xrd7 mro7 clk oro7 rq7 nrq7 1 0 DFLOPGC
Xra1 nrpGT1 nrq0 aro1 1 0 NAND2
Xro6 nrpGT1 aro1 oro6 1 0 NOR2
Xro7 oro6 cntClr oro7 1 0 NOR2

* Power oscillator
Xoa0 osc npwZ aoo0 1 0 NAND2
Xoa1 nosc npwZ aoo1 1 0 NAND2
Xoi0 osc nosc 1 0 INV
Xoi1 aoo0 p1a 1 0 INVx
Xoi2 aoo0 p1b 1 0 INVx
Xoi3 aoo1 p2a 1 0 INVx
Xoi4 aoo1 p2b 1 0 INVx

* Simulation Parameters
.TRAN .1n 2000n 0 1n

.END
```

Figure 134. Tactor Power Controller SPICE model source code. (continued)

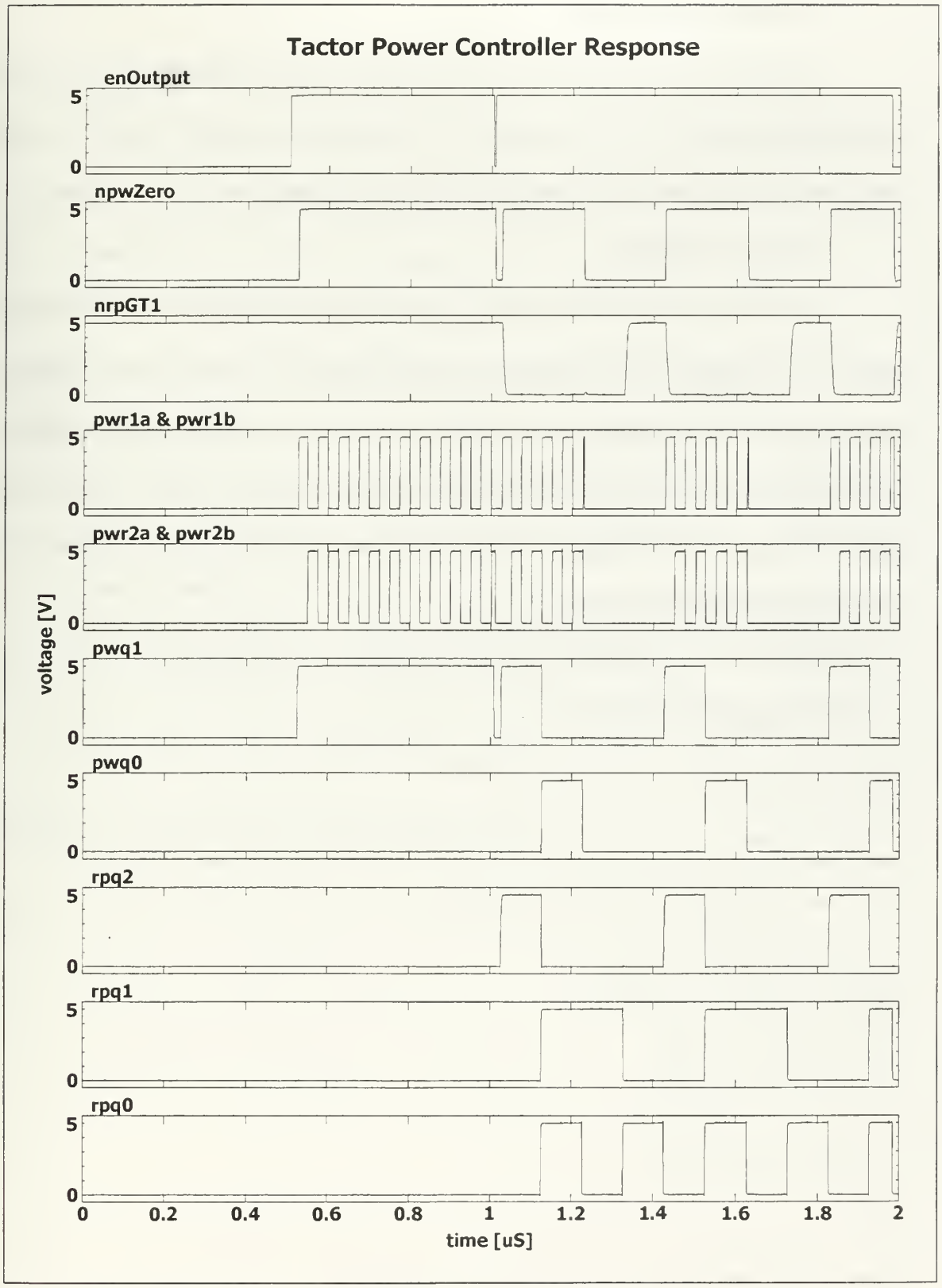


Figure 135. Tactor Power Controller SPICE model response.

1. Power Control Logic

The SPICE model for the power control logic structural design of Figure 106 was not individually tested since it is comprised of a single logic element. The power control logic component was tested as an integral portion of the Tactor Power Control module

2. Power Oscillator

The SPICE model for the power oscillator implements the structural design of Figure 107. A full source code listing is provided in Figure 136. An oscillation frequency is supplied to the power oscillator and the enable power signal is used to control transmission of the oscillation signal. The power oscillator response in Figure 137 shows that oscillation begins as soon as the enable power signal is applied and oscillation ends immediately when the enable power signal is removed.

```
* PwrOscill.cir ==>  Power Oscillator Transient Characteristics

* Logic Gate model definitions
.INCLUDE subckt.cir

* Power Supplies
VDS 1 0 5

* Input Signals
VenP enP 0 PWL(0 5 499n 5 500n 0 699n 0 700n 5 1499n 5 1500n 0 1699n 0
               1700n 5 1 5)
Vosc osc 0 PULSE(0 5 24.5n 1n 1n 24n 50n)

* Power oscillator
Xa0 osc enP ao0 1 0 NAND2
Xa1 nosc enP ao1 1 0 NAND2
Xi0 osc nosc 1 0 INV
Xi1 ao0 p1a 1 0 INVx
Xi2 ao0 p1b 1 0 INVx
Xi3 ao1 p2a 1 0 INVx
Xi4 ao1 p2b 1 0 INVx

* Simulation Parameters
.TRAN .1n 2000n 0 1n

.END
```

Figure 136. Power Oscillator SPICE model source code.

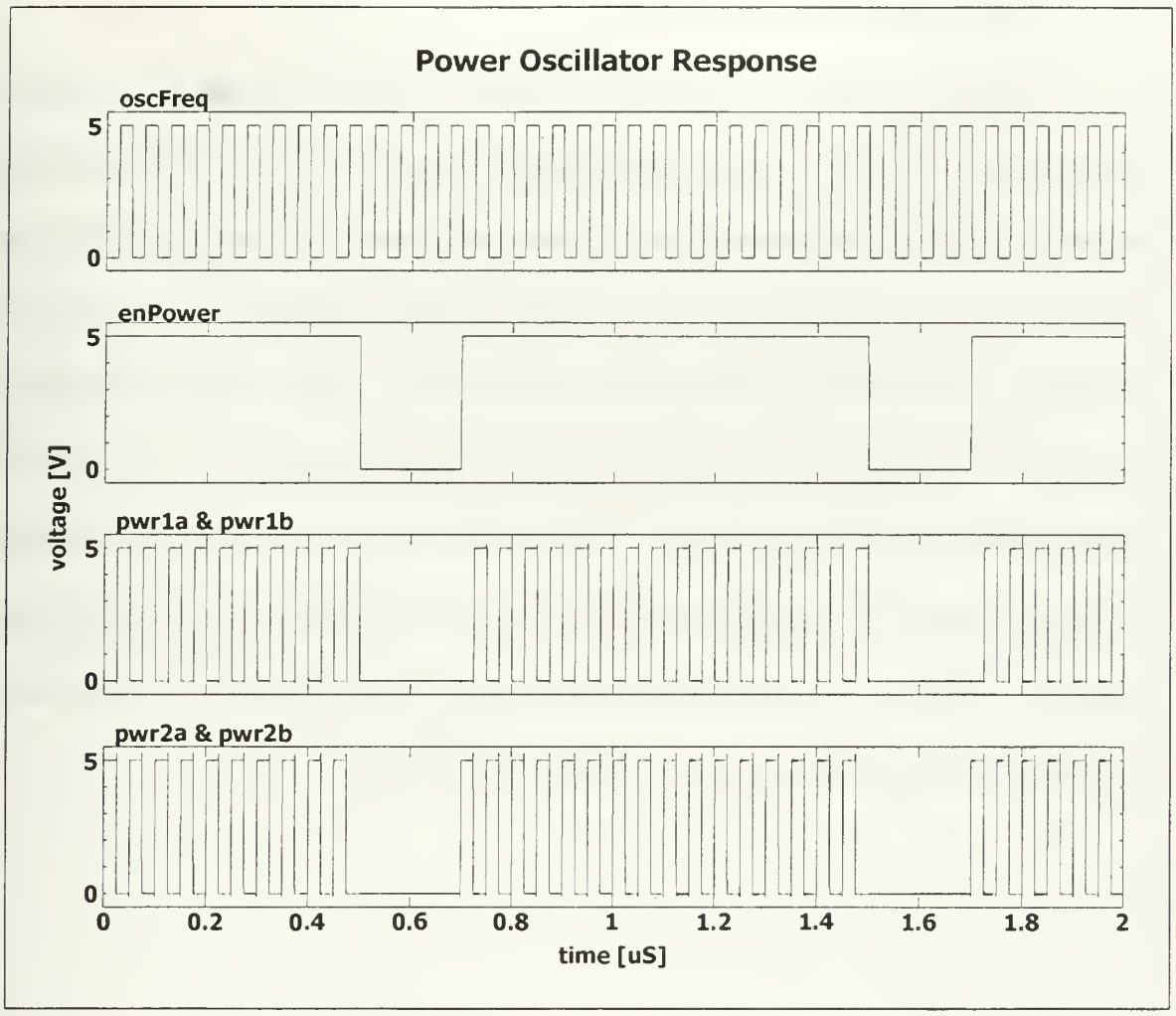


Figure 137. Power Oscillator SPICE model response.

3. Pulse Width Down Counter

The SPICE model for the pulse width down counter implements the structural design of Figure 108. A full source code listing is provided in Figure 138. Various values are loaded into the down counter to test the transitions between different stages. The pulse width down counter response in Figure 139 shows the values are loaded into the counter on the positive clock transition when the load signal is applied. The down counter decreases the stored value by one at each clock cycle. As seen just after the 1.2 μS point, when the counter reaches zero, it stops counting. The count clear signal at 1.7 μS immediately clears all counter stages. The control signal produced by this down counter is the "not pulse width equals zero." Figure 139 shows this flag is immediately applied whenever the count is non-zero and immediately cleared when the counter reaches zero.

```
* PWDownCount.cir ==>  Pulse Width Down Counter Transient Characteristics

* Logic Gate model definitions
.INCLUDE subckt.cir

* Power Supplies
VDS 1 0 5

* Input Signals
Vi0  i0  0 PWL(0 5 1 5)
Vi1  i1  0 PWL(0 0 1349n 0 1350n 5 1 5)
Vi2  i2  0 PWL(0 0 950n 0 951n 5 1250n 5 1251n 0 1349n 0 1350n 5 1 5)
Vi3  i3  0 PWL(0 0 650n 0 651n 5 950n 5 951n 0 1349n 0 1350n 5 1 5)
Vi4  i4  0 PWL(0 0 350n 0 351n 5 650n 5 651n 0 1349n 0 1350n 5 1 5)
Vi5  i5  0 PWL(0 5 350n 5 351n 0 1349n 0 1350n 5 1 5)
Vlod lod 0 PWL(0 0 49n 0 50n 5 99n 5 100n 0 349n 0 350n 5 399n 5 400n 0
               649n 0 650n 5 699n 5 700n 0 949n 0 950n 5 999n 5 1000n 0
               1349n 0 1350n 5 1399n 5 1400n 0 1 0)
Vclr clr 0 PWL(0 5 5n 5 6n 0 1709n 0 1710n 5 1719n 5 1720n 0 1 0)
Vclk clk 0 PULSE(0 5 24.5n 1n 1n 24n 50n)

* Pulse Width Down Counter
Xa0 oo0 oo1 npwZ 1 0 NAND2
Xo0 q3 q4 q5 oo0 1 0 NOR3
Xo1 q0 q1 q2 oo1 1 0 NOR3
Xm0 nq0 i0 lod mo0 1 0 MUX
Xd0 mo0 clk oo5 q0 nq0 1 0 DFLOPGC
Xx1 nq0 nq1 xo1 1 0 XNOR2
Xm1 xo1 i1 lod mo1 1 0 MUX
Xd1 mo1 clk oo5 q1 nq1 1 0 DFLOPGC
Xa1 nq0 nq1 ao1 1 0 NAND2
Xx2 ao1 q2 xo2 1 0 XNOR2
Xm2 xo2 i2 lod mo2 1 0 MUX
Xd2 mo2 clk oo5 q2 nq2 1 0 DFLOPGC
Xo2 ao1 q2 oo2 1 0 NOR2
Xx3 oo2 nq3 xo3 1 0 XNOR2
Xm3 xo3 i3 lod mo3 1 0 MUX
Xd3 mo3 clk oo5 q3 nq3 1 0 DFLOPGC
Xa2 oo2 nq3 ao2 1 0 NAND2
Xx4 ao2 q4 xo4 1 0 XNOR2
Xm4 xo4 i4 lod mo4 1 0 MUX
Xd4 mo4 clk oo5 q4 nq4 1 0 DFLOPGC
Xo3 ao2 q4 oo3 1 0 NOR2
Xx5 oo3 nq5 xo5 1 0 XNOR2
Xm5 xo5 i5 lod mo5 1 0 MUX
Xd5 mo5 clk oo5 q5 nq5 1 0 DFLOPGC
Xo4 lod npwZ oo4 1 0 NOR2
Xo5 oo4 clr oo5 1 0 NOR2

* Simulation Parameters
.TRAN .1n 2000n 0 1n

.END
```

Figure 138. Pulse Width Down Counter SPICE model source code.

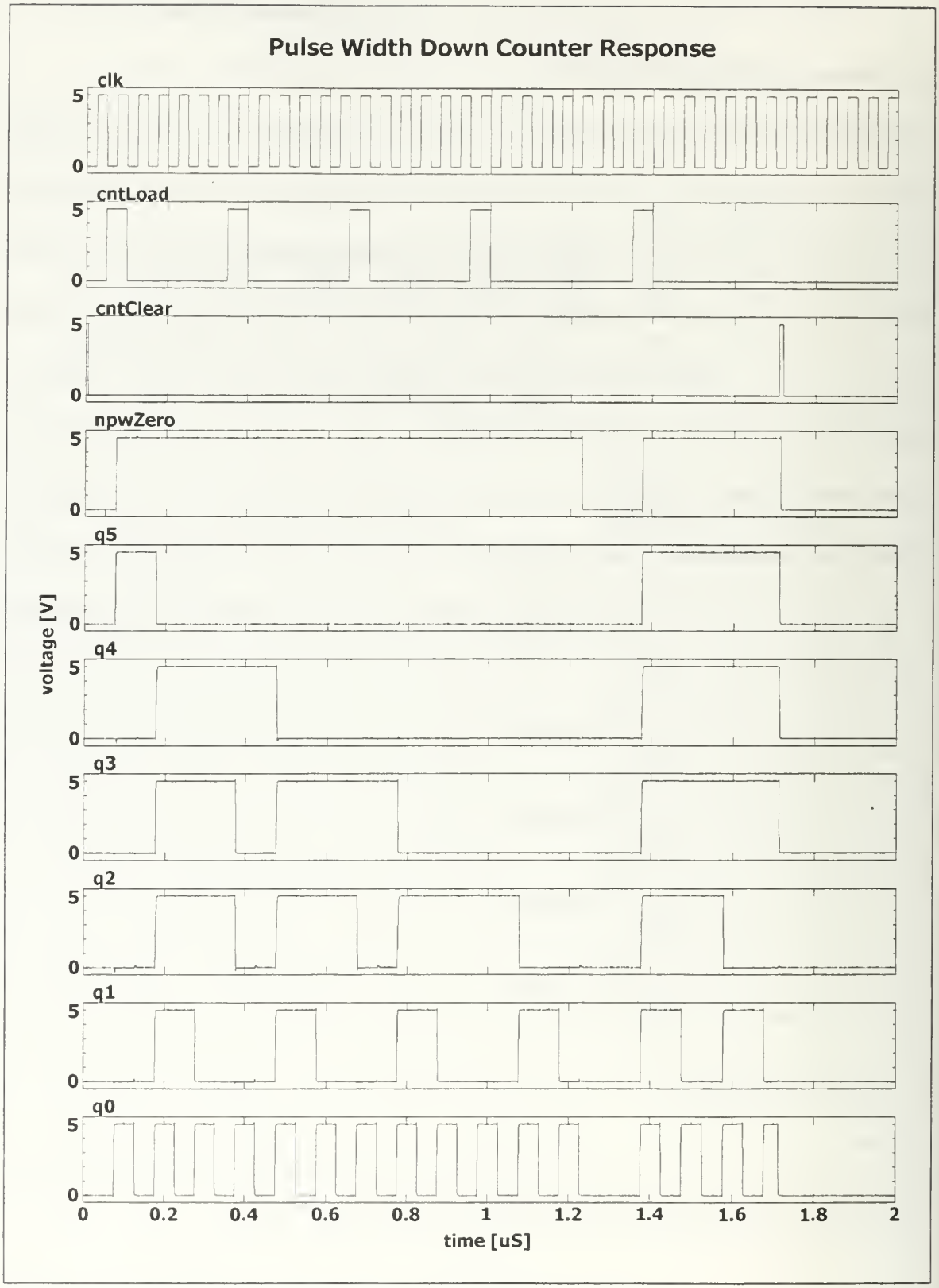


Figure 139. Pulse Width Down Counter SPICE model response.

4. Repetition Period Down Counter

The SPICE model for the repetition period down counter implements the structural design of Figure 109. A full source code listing is provided in Figure 140. Various values are loaded into the down counter to test the transitions between different stages. The repetition period down counter response in Figure 141 shows the values are loaded into the counter on the positive clock transition when the load signal is applied. The down counter decreases the stored value by one at each clock cycle. As seen just after the 1.6 μS point, when the counter reaches zero, it stops counting. The count clear signal at 1.2 μS immediately clears all counter stages. The control signal produced by this down counter is the "not repetition period greater than one." Figure 141 shows this flag is immediately applied whenever the count is one or zero and immediately cleared when the counter value is greater than one.

```
* RPDownCount.cir ==>  Repetition Period Down Counter Transient
Characteristics

* Logic Gate model definitions
.INCLUDE subckt.cir

* Power Supplies
VDS 1 0 5

* Input Signals
Vi0  i0  0 PWL(0 5 1 5)
Vi1  i1  0 PWL(0 0 1 0)
Vi2  i2  0 PWL(0 0 950n 0 951n 5 1250n 5 1251n 0 1 0)
Vi3  i3  0 PWL(0 0 650n 0 651n 5 950n 5 951n 0 1 0)
Vi4  i4  0 PWL(0 0 350n 0 351n 5 650n 5 651n 0 1 0)
Vi5  i5  0 PWL(0 5 350n 5 351n 0 1 0)
Vlod lod 0 PWL(0 0 49n 0 50n 5 99n 5 100n 0 349n 0 350n 5 399n 5 400n 0
               649n 0 650n 5 699n 5 700n 0 949n 0 950n 5 999n 5 1000n 0
               1349n 0 1350n 5 1399n 5 1400n 0 1 0)
Vclr clr 0 PWL(0 5 5n 5 6n 0 1179n 0 1180n 5 1189n 5 1190n 0 1 0)
Vclk clk 0 PULSE(0 5 24.5n 1n 1n 24n 50n)
```

Figure 140. Repetition Period Down Counter SPICE model source code.

```
* Repetition Period Down Counter
Xa0 oo0 oo1 oo2 nrpGT1 1 0 AND3
Xo0 q1 q2 oo0 1 0 NOR2
Xo1 q3 q4 oo1 1 0 NOR2
Xo2 q5 q6 q7 oo2 1 0 NOR3
Xm0 nq0 0 lod mo0 1 0 MUX
Xd0 mo0 clk oo7 q0 nq0 1 0 DFLOPGC
Xx1 nq0 nq1 xo1 1 0 XNOR2
Xm1 xo1 0 lod mo1 1 0 MUX
Xd1 mo1 clk oo7 q1 nq1 1 0 DFLOPGC
Xa2 nq0 nq1 ao2 1 0 NAND2
Xx2 ao2 q2 xo2 1 0 XNOR2
Xm2 xo2 i0 lod mo2 1 0 MUX
Xd2 mo2 clk oo7 q2 nq2 1 0 DFLOPGC
Xo3 ao2 q2 oo3 1 0 NOR2
Xx3 oo3 nq3 xo3 1 0 XNOR2
Xm3 xo3 i1 lod mo3 1 0 MUX
Xd3 mo3 clk oo7 q3 nq3 1 0 DFLOPGC
Xa3 oo3 nq3 ao3 1 0 NAND2
Xx4 ao3 q4 xo4 1 0 XNOR2
Xm4 xo4 i2 lod mo4 1 0 MUX
Xd4 mo4 clk oo7 q4 nq4 1 0 DFLOPGC
Xo4 ao3 q4 oo4 1 0 NOR2
Xx5 oo4 nq5 xo5 1 0 XNOR2
Xm5 xo5 i3 lod mo5 1 0 MUX
Xd5 mo5 clk oo7 q5 nq5 1 0 DFLOPGC
Xa4 oo4 nq5 ao4 1 0 NAND2
Xx6 ao4 q6 xo6 1 0 XNOR2
Xm6 xo6 i4 lod mo6 1 0 MUX
Xd6 mo6 clk oo7 q6 nq6 1 0 DFLOPGC
Xo5 ao4 q6 oo5 1 0 NOR2
Xx7 oo5 nq7 xo7 1 0 XNOR2
Xm7 xo7 i5 lod mo7 1 0 MUX
Xd7 mo7 clk oo7 q7 nq7 1 0 DFLOPGC
Xa1 nrpGT1 nq0 ao1 1 0 NAND2
Xo6 lod ao1 oo6 1 0 NOR2
Xo7 oo6 clr oo7 1 0 NOR2

* Simulation Parameters
.TRAN .1n 2000n 0 1n

.END
```

Figure 140. Repetition Period Down Counter SPICE model source code. (continued)

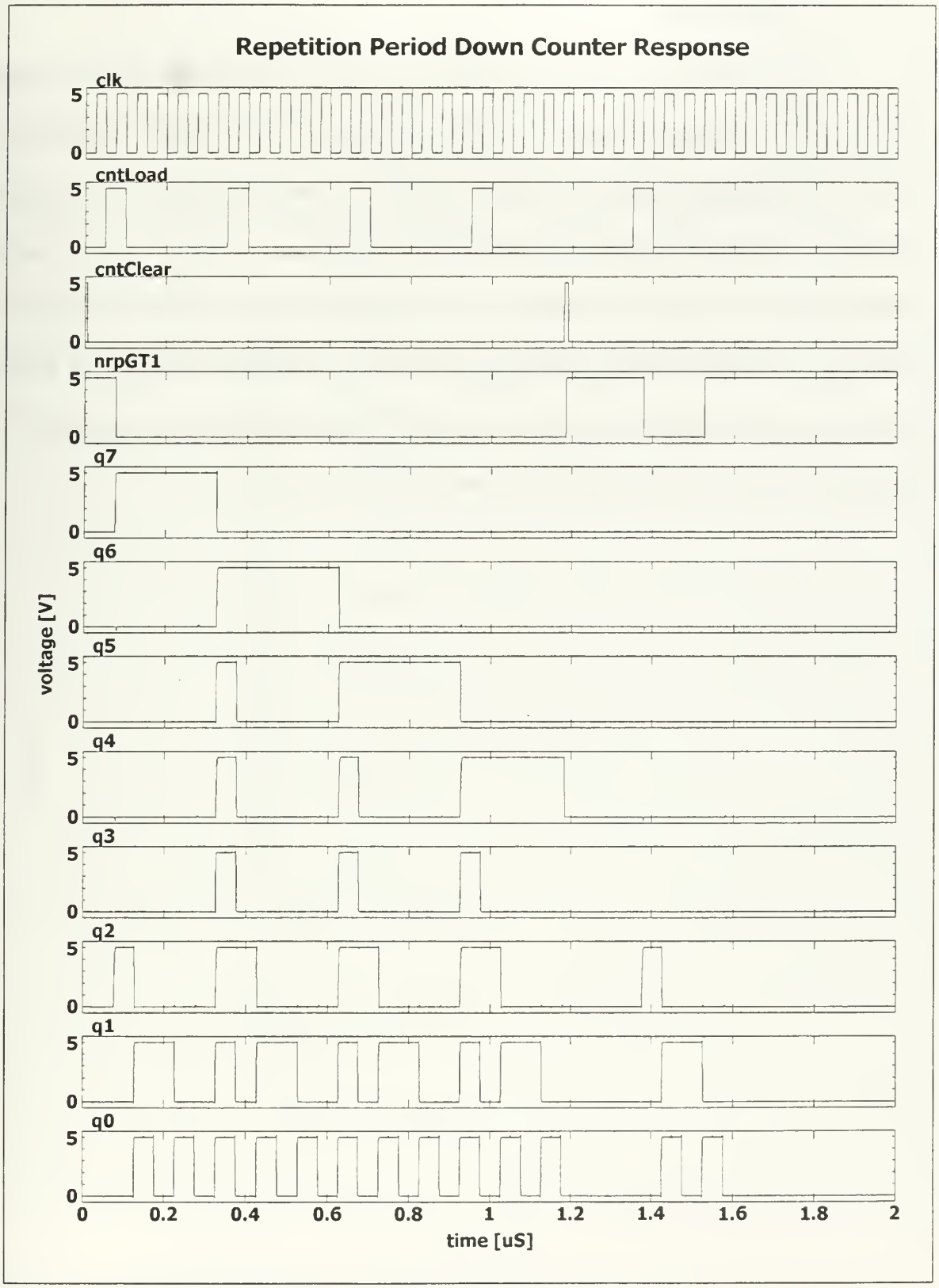


Figure 141. Repetition Period Down Counter SPICE model response.

5. Clock Divider

The SPICE model for the clock divider implements the structural design of Figure 110. A full source code listing is provided in Figure 142. The system clock drives a counter, causing the system clock to be divided by two at each counter stage. Clock divider testing was limited by the memory of the simulation computers. In order to test all fourteen stages of the clock divider, three different tests were required. Seven stages were tested at a time and a two-stage overlap was used to ensure that all broken connections were remade. The clock divider response in Figure 143 shows the results of the final component test. As required, each stage reduces the reference frequency by a factor of two.

```
* ClockDiv.cir ==>  Clock Divider Transient Characteristics

* Logic Gate model definitions
.INCLUDE subckt.cir

* Power Supplies
VDS 1 0 5

* Input Signals
Vnrst nRst 0 PWL(0 0 12n 0 13n 5 1849n 5 1850n 0 1899n 0 1900n 5 1 5)
Vclk clk 0 PULSE(0 5 24.5n 1n 1n 24n 50n)

* Clock Divider
Xd0 nq0 clk nRst q0 nq0 1 0 DFLOPGC
Xx1 q0 q1 xo1 1 0 XOR2
Xd1 xo1 clk nRst q1 nq1 1 0 DFLOPGC
Xa0 q0 q1 ao0 1 0 NAND2
Xx2 ao0 nq2 xo2 1 0 XOR2
Xd2 xo2 clk nRst q2 nq2 1 0 DFLOPGC
Xo0 ao0 nq2 oo0 1 0 NOR2
Xx3 oo0 q3 xo3 1 0 XOR2
Xd3 xo3 clk nRst q3 nq3 1 0 DFLOPGC
Xa1 oo0 q3 ao1 1 0 NAND2
Xx4 ao1 nq4 xo4 1 0 XOR2
Xd4 xo4 clk nRst q4 nq4 1 0 DFLOPGC
Xo1 ao1 nq4 oo1 1 0 NOR2
Xx5 oo1 q5 xo5 1 0 XOR2
Xd5 xo5 clk nRst q5 nq5 1 0 DFLOPGC
Xa2 oo1 q5 ao2 1 0 NAND2
Xx6 ao2 nq6 xo6 1 0 XOR2
Xd6 xo6 clk nRst q6 nq6 1 0 DFLOPGC
Xo2 ao2 nq6 oo2 1 0 NOR2
Xx7 oo2 q7 xo7 1 0 XOR2
Xd7 xo7 clk nRst q7 nq7 1 0 DFLOPGC
Xa3 oo2 q5 ao3 1 0 NAND2
Xx8 ao3 nq8 xo8 1 0 XOR2
Xd8 xo8 clk nRst q8 nq8 1 0 DFLOPGC
Xo3 ao3 nq6 oo3 1 0 NOR2
Xx9 oo3 q9 xo9 1 0 XOR2
Xd9 xo9 clk nRst q9 nq9 1 0 DFLOPGC
Xa4 oo3 q5 ao4 1 0 NAND2
Xx10 ao4 nq10 xo10 1 0 XOR2
Xd10 xo10 clk nRst q10 nq10 1 0 DFLOPGC
Xo4 ao4 nq6 oo4 1 0 NOR2
Xx11 oo4 q11 xo11 1 0 XOR2
Xd11 xo11 clk nRst q11 nq11 1 0 DFLOPGC
Xa5 oo4 q5 ao5 1 0 NAND2
Xx12 ao5 nq12 xo12 1 0 XOR2
Xd12 xo12 clk nRst q12 nq12 1 0 DFLOPGC
Xo5 ao5 nq6 oo5 1 0 NOR2
Xx13 oo5 q13 xo13 1 0 XOR2
Xd13 xo13 clk nRst q13 nq13 1 0 DFLOPGC

* Simulation Parameters
.TRAN .1n 2000n 0 1n

.END
```

Figure 142. Clock Divider SPICE model source code.

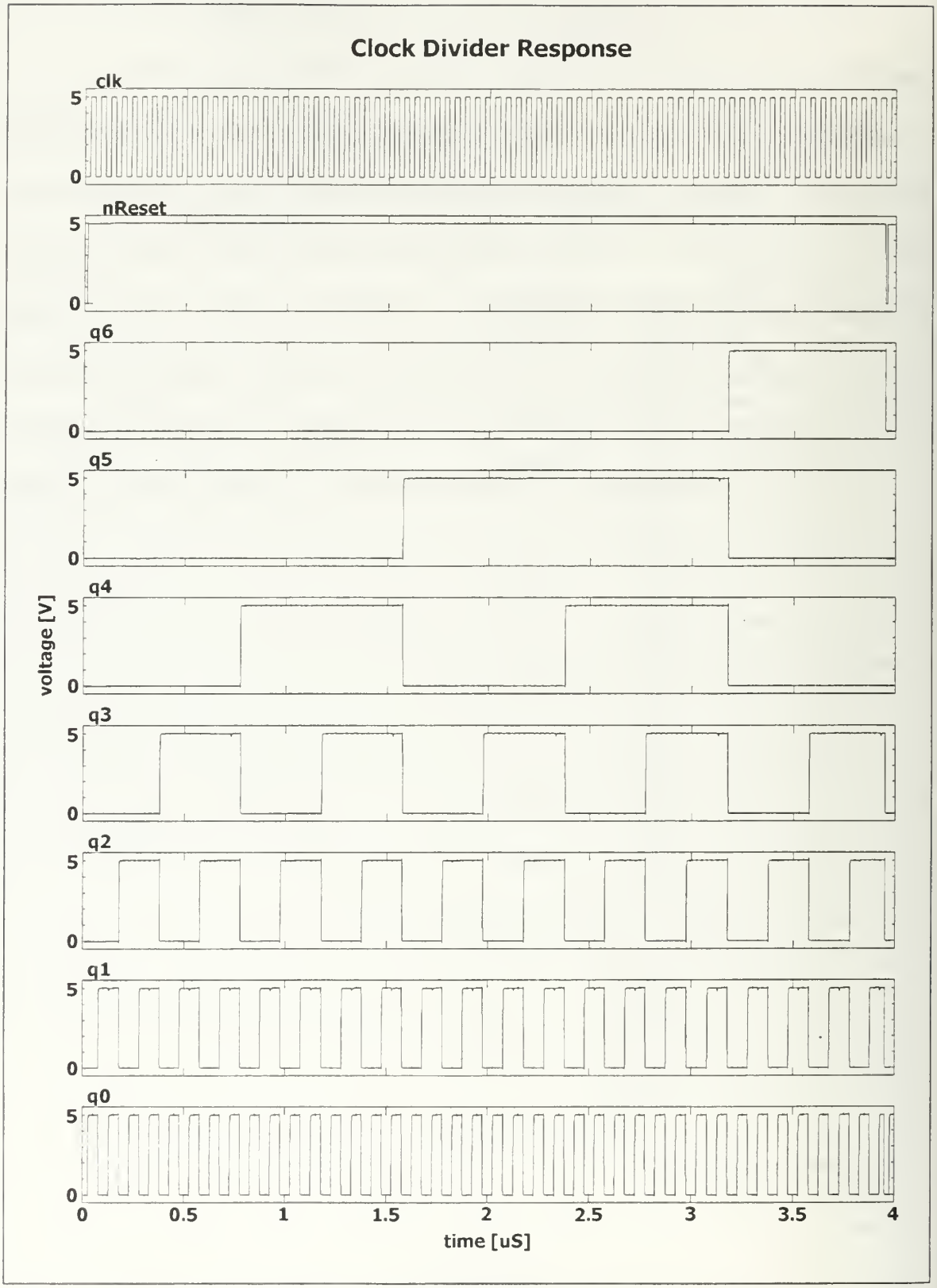


Figure 143. Clock Divider SPICE model response.

APPENDIX D. TACTILE INTERFACE ANIMATION PROGRAM

The goal of animating TIC operations was to accurately and clearly portray the functional relations between the wave controlling components. An animation was developed that illustrates the changes that occur in the TIC registers and counters in response to a series of command bytes.

A. ANIMATION DESIGN

1. TIC Visual Representation

The significant TIC changes caused by received commands include the controller state, both wave shape registers, and both down counters. It was essential to depict tactor activation as a visual vibration of the tactor because this system uses vibration as the physical stimulus. Figure 144 shows the graphical representation of two intelligent tactors in a tactile array. The dark gray rectangles represent the tactors. Each is labeled with its address value. Two labeled columns are provided for each tactor to depict the parameters associated with pulse width and repetition period. The number at the bottom of each column is the stored register value for the pulse width or repetition period. The column acts as a vertical gage representing the value in the down counter associated with each register. The horizontal bar across the bottom indicates the simulation time and proceeds steadily from left to right. The rectangular bubbles above the time line are commands that will be issued when the time reaches their position.

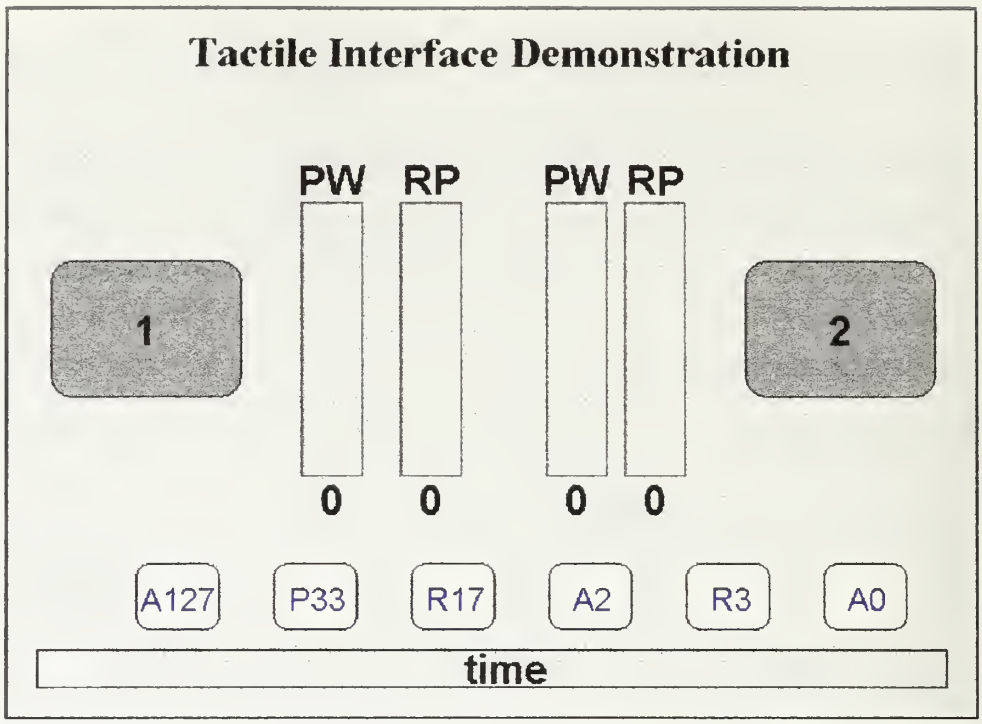


Figure 144. Tactile Interface Animation Elements.

2. Animation Color Scheme

A color scheme was conceived to convey additional information regarding TIC conditions. The tactor rectangles change in color to represent the state of the command sequence controller. The color used for to fill the pulse width gage is green, exhibiting a "go" condition for tactor activation any time the count is greater than zero.

Figure 145 shows the animation in progress. When a valid address is received the TIC shifts to state "B" and the tactor color changes to yellow. When a register command is received by a tactor in state "B," the register is set to the commanded value, the TIC shifts to state "C," and the tactor color changes to red. When any address is received by a tactor in state "C," the TIC shifts to state "A" and the tactor color changes back to gray. The repetition period down counter value is represented by a blue column in the area below the "RP" label. During operation, the pulse width gage falls four times as fast as the repetition

period gage. When the pulse width column is not zero, the attached tactor vibrates. When the pulse width column reaches zero, vibration stops. When the repetition period is less than two, both down counters load the stored register value. Consequently, when a zero repetition period is assigned, both columns reload on every clock pulse and neither column decreases in value. When the repetition period register is greater than zero, both counters decrease until they are reloaded at the repetition period down counter value of one.

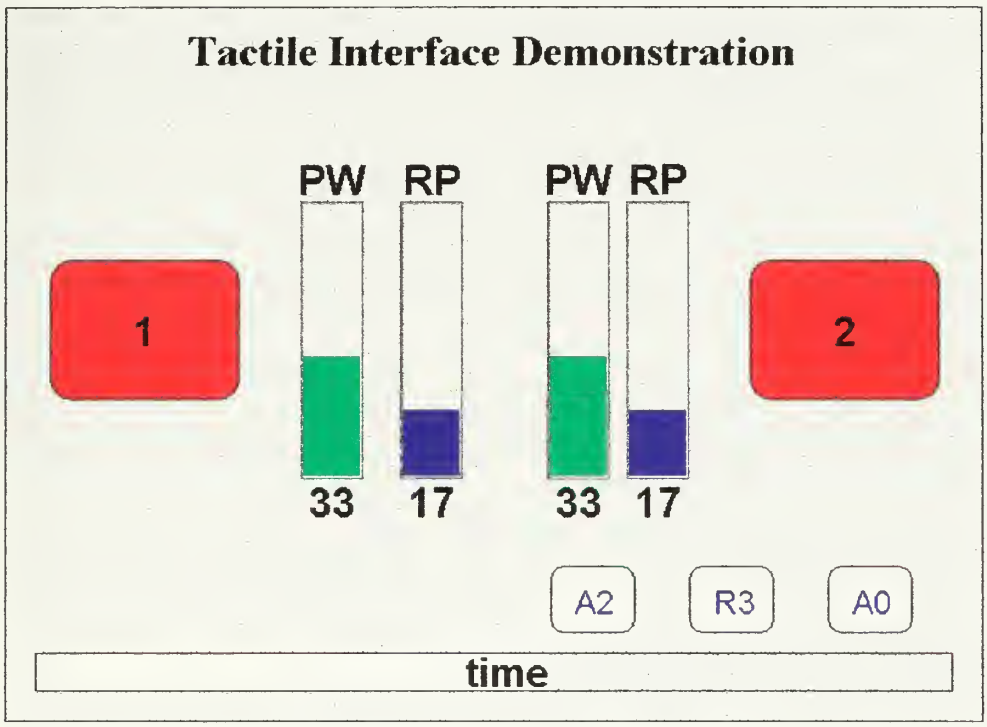


Figure 145. Tactile Interface Animation in Progress.

B. ANIMATION PROGRAMMING

Initially, C++ was used to develop the TIC animation. This choice was a mistake due to the complexity of C++ programming with respect to event timing and graphics display. Programming efforts were shifted to making a JAVA applet that would run in a web browser because this application is very limited in scope. With a score of demonstration applets and extensive documentation, the JAVA implementation was much

easier that the C++ effort. The program was divided into three logical objects: the intelligent tactors, the command bytes, and the demonstration events. Each of these elements was implemented as an object class and they are discussed in the following subsections.

1. Intelligent Tactor

The "Tactor" class maintains all required parameters for intelligent tactor simulation. This object includes methods for initialization, command reception, and graphic display. Figure 146 contains the complete JAVA source code that implements the tactor class.

```
/* ---------------------------------------------------------------------
 * Tactor 1.1 ==> This class creates and manages Intelligent Tactors.
 *
 * Copyright (c) 1999 Jeff Link, All Rights Reserved.
 * Permission to use, copy, modify, and distribute this software and its
 * documentation for NON-COMMERCIAL purposes and without fee is hereby
 * granted.
 *
 * The author makes on claims regarding the suitability of this software
 * and shall not be liable for any damages suffered as a result of using,
 * modifying, or distributing this software or its derivatives.
 * ---------------------------------------------------------------------*/

import java.awt.Graphics;
import java.awt.Color;
import java.awt.Font;

public class Tactor {
  private int cX, cY, direction;
  private int address, state, pwReg, rpReg;
  private int pwCount, rpCount, active;
  private Color pwFill,rpFill,border;
  private Color[] clrState = new Color[3];
  private Font labelFont = new Font("Serif", Font.BOLD, 30);
  private String strAddr,strPWReg,strRPReg;
```

Figure 146. Intelligent Tactor object JAVA source code.

```
/**
 * Constructs a Tactor.
 * @param in_centerx       The x-coord of the center
 * @param in_centery       The y-coord of the center
 * @param in_direction     The direction 0 = Left, 1 = Right
 * @param in_addree        Assigned address: 1 - 126
 */
public Tactor(int in_centerx, int in_centery, int in_direction,
              int in_address) {
  clrState[0] = Color.gray;
  clrState[1] = Color.yellow;
  clrState[2] = Color.red;
  border=Color.black;
  pwFill=Color.green;
  rpFill=Color.blue;
  cX=in_centerx;
  cY=in_centery;
  direction=in_direction;
  address=in_address;
  strAddr=String.valueOf(address);
  initialize();
}

/**
 * Initializes TIC values.
 * @param none
 */
public void initialize() {
  state = 0;
  pwCount=pwReg=0;
  rpCount=rpReg=0;
  strPWReg=String.valueOf(pwReg);
  strRPReg=String.valueOf(rpReg);
}
```

Figure 146. Intelligent Tactor object JAVA source code. (continued)

```
/**
 * Sends a command to the TIC.
 * @param cmd
 */
public void issueCommand(int cmd) {
  switch (state) {
    case 0:
      if (cmd==address || cmd==127)
        state=1;
      break;
    case 1:
      if (cmd>127)
        state=2;
      if (cmd>127 && cmd<192) {
        pwReg=pwCount=cmd-128;
        strPWReg=String.valueOf(pwReg);
        rpCount=4*rpReg;
      }
      if (cmd>191 && cmd<256) {
        rpReg=cmd-192;
        rpCount=4*rpReg;
        strRPReg=String.valueOf(rpReg);
        pwCount=pwReg;
      }
      break;
    case 2:
      if (cmd>=0 && cmd<128)
        state=0;
      if (cmd==address || cmd==127)
        state=1;
      if (cmd>127 && cmd<192) {
        pwReg=pwCount=cmd-128;
        strPWReg=String.valueOf(pwReg);
        rpCount=4*rpReg;
      }
      if (cmd>191 && cmd<256) {
        rpReg=cmd-192;
        rpCount=4*rpReg;
        strRPReg=String.valueOf(rpReg);
        pwCount=pwReg;
      }
      break;
  }
}
```

Figure 146. Intelligent Tactor object JAVA source code. (continued)

```
  /**
   * Updates the TIC parameters.
   * @param
   */
  public void updateTactor(int ticks) {
    if (pwCount>0)
      --pwCount;
    if (rpCount>1)
      --rpCount;
    else {
      pwCount=pwReg;
      rpCount=4*rpReg;
    }
    if (pwCount>0)
      active=(ticks%2==0?1:-1);
    else
      active=0;
  }

  /**
   * Draws the tactor on a graphics object.
   * @param g       The graphics object which the tactor will be drawn upon.
   */
  public void drawTactor(Graphics g) {
    int tX=cX+(direction>0?15:-150)-active;
    int tY=cY-54+active;
    int dX=138+2*active;
    int dY=99-2*active;
    g.setColor(clrState[state]);
    g.fillRoundRect(tX,tY,dX+1,dY+1,30,30);
    g.setColor(Color.black);
    g.drawRoundRect(tX,tY,dX,dY,30,30);
    g.setFont(labelFont);
    g.drawString("PW",cX+(direction>0?-132:30),cY-102);
    g.drawString("RP",cX+(direction>0?-71:106),cY-102);
    g.drawString(strPWReg,cX+(direction>0?-132:30)+charOffset(pwReg),cY+132);
    g.drawString(strRPReg,cX+(direction>0?-75:102)+charOffset(rpReg),cY+132);
    g.drawString(strAddr,cX+(direction>0?60:-105)+charOffset(address),cY+8);

    g.setColor(Color.darkGray);
    g.drawRect(cX+(direction>0?-73:32),cY-97,44,200);
    g.drawRect(cX+(direction>0?-130:104),cY-97,44,200);
    g.setColor(Color.lightGray);
    g.draw3DRect(cX+(direction>0?-72:33),cY-96,42,198,false);
    g.draw3DRect(cX+(direction>0?-129:105),cY-96,42,198,false);
    g.setColor(pwFill);
    g.fillRect(cX+(direction>0?-128:34),cY+102-3*pwCount,41,3*pwCount);
    g.setColor(rpFill);
    g.fillRect(cX+(direction>0?-71:106),cY+102-(int)(3*rpCount/4),41,
                                                    (int)(3*rpCount/4));
  }
```

Figure 146. Intelligent Tactor object JAVA source code. (continued)

```
  private int charOffset(int iTmp) {
    int iRtn=0;
    if (iTmp<100)
      ++iRtn;
    if (iTmp<10)
      ++iRtn;
    return iRtn*8;
  }
}
```

Figure 146. Intelligent Tactor object JAVA source code. (continued)

2. Command Byte

The "TIC Command" class contains the command byte and transmission time values for each simulation command. This object includes methods for initialization, parameter retrieval, and graphic display. Figure 147 contains the complete JAVA source code that implements the TIC command class.

```
/* ------------------------------------------------------------------
 * TICCommand 1.1 ==> This class manages commands to be sent to the TIC.
 *
 * Copyright (c) 1999 Jeff Link, All Rights Reserved.
 * Permission to use, copy, modify, and distribute this software and its
 * documentation for NON-COMMERCIAL purposes and without fee is hereby
 * granted.
 *
 * The author makes on claims regarding the suitability of this software
 * and shall not be liable for any damages suffered as a result of using,
 * modifying, or distributing this software or its derivatives.
 * ------------------------------------------------------------------*/

import java.awt.Graphics;
import java.awt.Color;
import java.awt.Font;
import java.lang.Math;

public class TICCommand {
  private int word, time;
  private Color clrText, clrBorder, clrBack;
  private Font labelFont = new Font("Serif", Font.PLAIN, 24);
  private String strWord;
```

Figure 147. TIC Command object JAVA source code.

```
  /**
   * Constructs a TICCommand.
   * @param in_word     The byte command to transmit
   * @param in_time     Time to transmit the command
   */
  public TICCommand(int in_word, int in_time) {
    clrText = Color.blue;
    clrBorder = Color.black;
    clrBack = Color.white;
    word=in_word;
    time=in_time;
    if (word<128)
      strWord="A"+String.valueOf(word);
    else if (word<192)
      strWord="P"+String.valueOf(word-128);
    else
      strWord="R"+String.valueOf(word-192);
  }

  /**
   * Gets the command word.
   */
  public int getCommand() {
    return word;
  }

  /**
   * Gets the time to transmit.
   */
  public int getTime() {
    return time;
  }

  /**
   * Draws the TICCommand on a graphics object.
   * @param g           The graphics object to draw the command upon
   * @param cX          The x-coord for command center
   * @param cY          The y-coord for command center
   */
  public void drawCommand(Graphics g, int cX, int cY) {
    int tX=cX-8, tY=cY-23, dX=60, dY=48;
    g.setColor(clrBack);
    g.fillRoundRect(tX,tY,dX+1,dY+1,20,20);
    g.setColor(clrBorder);
    g.drawRoundRect(tX,tY,dX,dY,20,20);
    g.setColor(clrText);
    g.setFont(labelFont);
    g.drawString(strWord,tX+32-strWord.length()*7,tY+35);
  }
}
```

Figure 147. TIC Command object JAVA source code. (continued)

3. TIC Demo

The "TIC Demo" class contains the parameters required to combine the tactor and TIC command classes into an animated tactile array. This class is the core of the applet and implements multi-threading and mouse event processing. The TIC demonstration instantiates two intelligent tactors and an array of TIC commands. This object includes methods for initialization, mouse event processing, animation timing, and graphic display. The applet continuously loops through time, resetting to zero when the maximum time is reached. The applet resets to the initial conditions when the mouse button is released in the applet active area. Figure 148 contains the complete JAVA source code that implements the TIC demonstration class.

```
/* ------------------------------------------------------------------
 * TICDemo 1.1 ==> This class demonstrates TIC operations.
 *
 * Copyright (c) 1999 Jeff Link, All Rights Reserved.
 * Permission to use, copy, modify, and distribute this software and its
 * documentation for NON-COMMERCIAL purposes and without fee is hereby
 * granted.
 *
 * The author makes on claims regarding the suitability of this software
 * and shall not be liable for any damages suffered as a result of using,
 * modifying, or distributing this software or its derivatives.
 * ------------------------------------------------------------------*/

import java.awt.Graphics;
import java.awt.Color;
import java.awt.Image;
import java.awt.Font;
import java.awt.event.MouseListener;
import java.awt.event.MouseEvent;
import Tactor;
import TICCommand;
```

Figure 148. Tactile Array Demonstration object JAVA source code.

```
public class TICDemo extends java.applet.Applet
                     implements Runnable, MouseListener {
  private int sleep=100,height=440,width=700,tt,currCmd,lastCmd;
  private Tactor t1 = new Tactor(180, 170, 0, 1);
  private Tactor t2 = new Tactor(520, 170, 1, 2);
  private Thread animate=null;
  Image backBuffer;
  private Graphics backGC;
  private TICCommand[] cmdList;
  private Font labelFont = new Font("Serif", Font.BOLD, 30);

  public void init() {       // Initialize all variables and classes
    tt=0;
    fillCommands();
    updateTactors();
    try {
      backBuffer = createImage(width, height);
      backGC = backBuffer.getGraphics();
    } catch (Exception e) { backGC=null; }
    addMouseListener(this);
  }

  public void destroy() {    // Class destructor
    removeMouseListener(this);
  }

  private void fillCommands() {   // Fill command array
    cmdList = new TICCommand[10];
    cmdList[0] = new TICCommand(127,100);
    cmdList[1] = new TICCommand(161,200);
    cmdList[2] = new TICCommand(209,300);
    cmdList[3] = new TICCommand(2,400);
    cmdList[4] = new TICCommand(195,500);
    cmdList[5] = new TICCommand(0,600);
    currCmd=0;
    lastCmd=5;
    cmdList[6] = new TICCommand(0,0);
  }

  private void updateTactors() {  // Updates the tactors when required
    if (tt==0) {   // reset tactors and command list when time restarts
      t1.initialize();
      t2.initialize();
      currCmd=0;
    }
    if (tt==cmdList[currCmd].getTime()) {   // transmit the command
      t1.issueCommand(cmdList[currCmd].getCommand());
      t2.issueCommand(cmdList[currCmd].getCommand());
      ++currCmd;
    }
    t1.updateTactor(tt);
    t2.updateTactor(tt);
  }
```

Figure 148. Tactile Array Demonstration object JAVA source code. (continued)

```
  private void paintApplet(Graphics g) {  // Paint the applet
    int ii=currCmd;
    t1.drawTactor(g);
    t2.drawTactor(g);
    while (ii <= lastCmd) {
      cmdList[ii].drawCommand(g, cmdList[ii].getTime(), 360);
      ++ii;
    }
  }

  public void update(Graphics g) { // When update is called
    if (backBuffer != null) {
      // double-buffering available
      backGC.setColor(Color.lightGray);
      backGC.fillRect(0,0,width,height);
      backGC.setColor(Color.white);
      backGC.fillRect(20,400,tt+1,27);
      backGC.setColor(Color.black);
      backGC.drawRect(20,400,661,27);
      backGC.setFont(labelFont);
      backGC.drawString("time",width/2-20,425);
      paintApplet(backGC);
      g.drawImage(backBuffer, 0, 0, this);
    }
    else {
      // no double-buffering
      g.setColor(Color.lightGray);
      g.fillRect(0,0,width,height);
      g.setColor(Color.white);
      g.fillRect(20,400,tt+1,27);
      g.setColor(Color.black);
      g.drawRect(20,400,661,27);
      g.setFont(labelFont);
      g.drawString("time",width/2-20,425);
      paintApplet(g);
    }
  }

  public void run() { //Run the applet
    while (true) {
      if (tt<660)
        ++tt;
      else
        tt=0;
      updateTactors();
      repaint();
      try { Thread.sleep(sleep); } catch (InterruptedException e) { }
    }
  }

  public void start() {  // When the applet is started
    if (animate == null) {
      animate = new Thread(this);
      animate.start();
    }
  }
```

Figure 148. Tactile Array Demonstration object JAVA source code. (continued)

```
  public void stop() {  // When the applet is stopped
    if (animate != null)
      animate=null;
  }

  // These functions are required for the MouseListener interface

  public void mouseReleased(MouseEvent e) { // Clicked on demo
    tt=0;
    updateTactors();
    repaint();
  }

  public void mousePressed(MouseEvent e) { }
  public void mouseEntered(MouseEvent e) { }
  public void mouseExited(MouseEvent e) { }
  public void mouseClicked(MouseEvent e) { }
}
```

Figure 148. Tactile Array Demonstration object JAVA source code. (continued)

4. Demonstration Applet HTML File

A JAVA applet runs as a process embedded in an "html" file. This configuration allows applets to be executed by any JAVA compliant web browser. Figure 149 contains the complete HTML code that executes the Tactile Interface demonstration applet.

```
<html>
<head>
<meta http-equiv="Content-Type"
content="text/html; charset=iso-8859-1">
<title>Tactile Interface Demonstration (1.1)</title>
</head>

<body bgcolor="#C0C0C0">

<h1 align="center">Tactile Interface Demonstration</h1>

<p align="center"><applet code="TICDemo.class" codebase="./"
align="baseline" width="700" height="440"></applet> </p>
</body>
</html>
```

Figure 149. Tactile Demonstration Applet HTML source code.

APPENDIX E. VLSI LOGIC ELEMENT DESIGN

VLSI logic element design consists of determining the schematic transistor connections, testing the planned circuit using SPICE simulation, and then constructing the element using the VLSI layers. This appendix provides those three design phases for all logic elements in the TIC design.

A. LOGIC ELEMENT SCHEMATICS

The schematics for all logic elements are contained in the following subsections.

1. Inverter

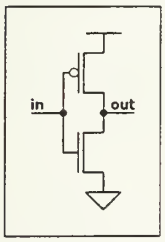


Figure 150. Inverter schematic.

2. Two Input NAND

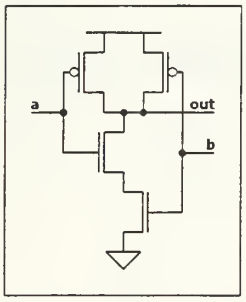


Figure 151. Two Input NAND Gate schematic.

3. Three Input NAND

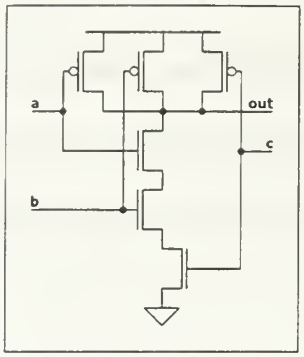


Figure 152. Three Input NAND Gate schematic.

4. Four Input NAND

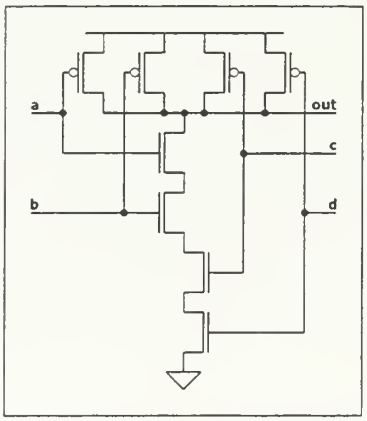


Figure 153. Four Input NAND Gate schematic.

5. Three Input AND

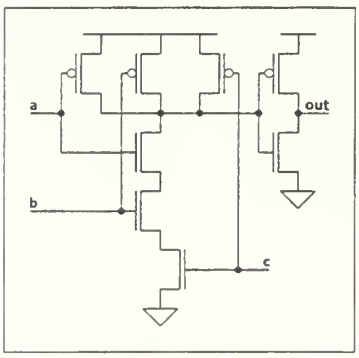


Figure 154. Three Input AND Gate schematic.

6. Four Input AND

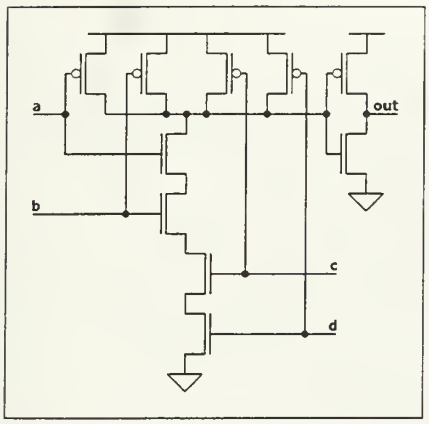


Figure 155. Four Input AND Gate schematic.

7. Two Input NOR

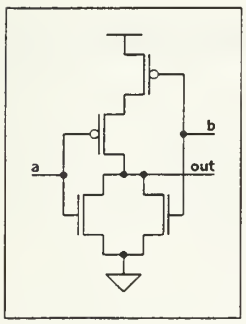


Figure 156. Two Input NOR Gate schematic.

8. Three Input NOR

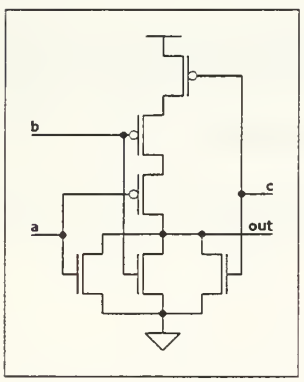


Figure 157. Three Input NOR Gate schematic.

9. Two Input XOR

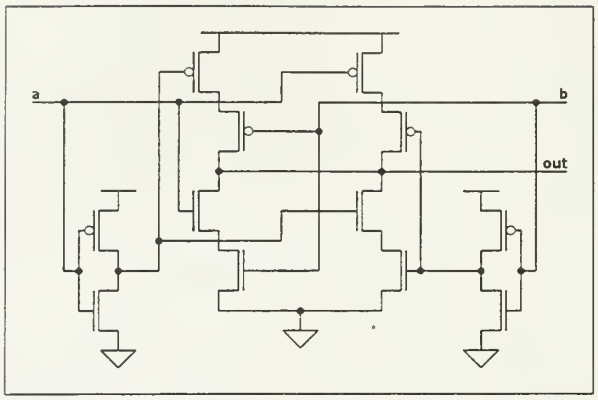


Figure 158. Two Input XOR Gate schematic.

10. Two Input XNOR

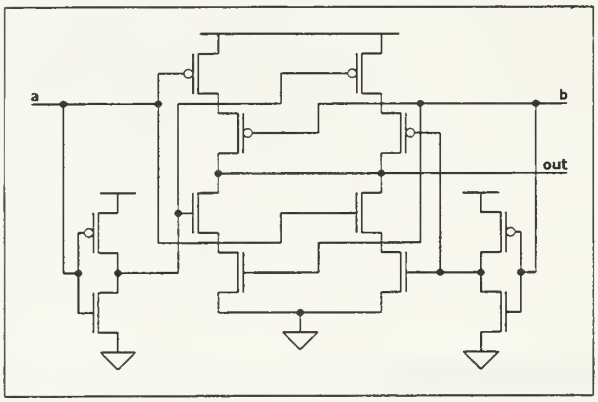


Figure 159. Two Input XNOR Gate schematic.

11. D Flip Flop with Clear

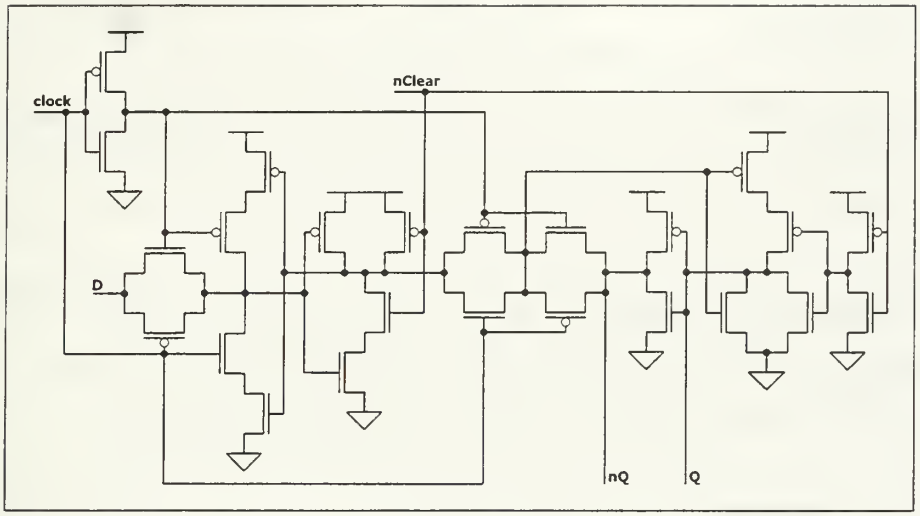


Figure 160. D Flip Flop with Clear schematic.

12. Two Input Multiplexer

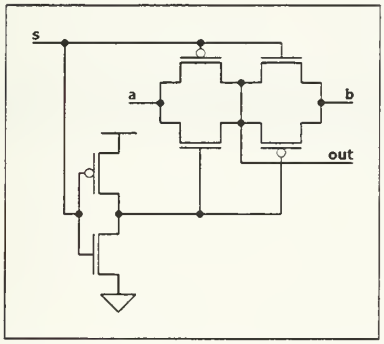


Figure 161. Two Input Multiplexer schematic.

B. LOGIC ELEMENT SPICE SIMULATIONS

The SPICE simulation files used to evaluate logic element design response are contained in the following subsections.

1. Inverter

```
* File: inverter.cir
* CMOS Inverter DC Transfer Characteristics

* CMOSP & CMOSN model definitions
.INCLUDE cmos.cir
.INCLUDE subckt.cir

* Power Supplies
VDS 1 0 5

* Input Signals
VINa a 0 PULSE(0 5 .5N 1N 1N 19N 40N)

* Main Circuit
Xia a o 1 0 INV

C1 o 0 .1P

* Simulation Parameters
.TRAN .001N 40N

.END
```

Figure 162. Inverter SPICE model source code.

2. Two Input NAND

```
* File: nand2.cir
* CMOS 2-input NAND Transient Characteristics

* CMOSP & CMOSN model definitions
.INCLUDE cmos.cir
.INCLUDE subckt.cir

* Power Supplies
VDS 1 0 5

* Input Signals
VINa a 0 PWL(0n 5 10n 5 11n 0 20n 0 21n 5 30n 5 31n 0 40n 0 41n 5 50n 5 51n 5
             60n 5 61n 5 70n 5 71n 0 80n 0 81n 0 90n 0 91n 0 100n 0 101n 5
             110n 5 111n 0 120n 0 121n 5 130n 5 131n 0 140n 0)
VINb b 0 PWL(0n 5 10n 5 11n 0 20n 0 21n 5 30n 5 31n 5 40n 5 41n 5 50n 5 51n 0
             60n 0 61n 5 70n 5 71n 0 80n 0 81n 5 90n 5 91n 0 100n 0 101n 0
             110n 0 111n 5 120n 5 121n 0 130n 0 131n 0 140n 0)

* Main Circuit
Xla a b o 1 0 NAND2

C1 o 0 .1P

* Simulation Parameters
.TRAN .001N 140N

.END
```

Figure 163. Two Input NAND gate SPICE model source code.

3. Three Input NAND

```
* File: nand3.cir
* CMOS 3-input NAND Transient Characteristics

* CMOSP & CMOSN model definitions
.INCLUDE cmos.cir
.INCLUDE subckt.cir

* Power Supplies
VDS 1 0 5

* Input Signals
VINa a 0 PWL(0n 5 10n 5 11n 5 20n 5 21n 5 30n 5 31n 5 40n 5 41n 5 50n 5 51n 0
             60n 0 61n 5 70n 5)
VINb b 0 PWL(0n 5 10n 5 11n 5 20n 5 21n 5 30n 5 31n 0 40n 0 41n 5 50n 5 51n 5
             60n 5 61n 5 70n 5)
VINc c 0 PWL(0n 5 10n 5 11n 0 20n 0 21n 5 30n 5 31n 5 40n 5 41n 5 50n 5 51n 5
             60n 5 61n 5 70n 5)

* Main Circuit
Xla a b c o 1 0 NAND3

C1 o 0 .1P

* Simulation Parameters
.TRAN .0001N 70N

.END
```

Figure 164. Three Input NAND gate SPICE model source code.

4. Four Input NAND

```
* File: nand4.cir
* CMOS 4-input NAND Transient Characteristics

* CMOSP & CMOSN model definitions
.INCLUDE cmos.cir

* Power Supplies
VDS 1 0 5

* Input Signals
VINa a 0 PWL(0n 5 10n 5 11n 5 20n 5 21n 5 30n 5 31n 5 40n 5 41n 5 50n 5 51n 0
             60n 0 61n 5 70n 5 71n 5 80n 5 81n 5 90n 5)
VINb b 0 PWL(0n 5 10n 5 11n 5 20n 5 21n 5 30n 5 31n 0 40n 0 41n 5 50n 5 51n 5
             60n 5 61n 5 70n 5 71n 5 80n 5 81n 5 90n 5)
VINc c 0 PWL(0n 5 10n 5 11n 0 20n 0 21n 5 30n 5 31n 5 40n 5 41n 5 50n 5 51n 5
             60n 5 61n 5 70n 5 71n 5 80n 5 81n 5 90n 5)
VINd d 0 PWL(0n 5 10n 5 11n 5 20n 5 21n 5 30n 5 31n 5 40n 5 41n 5 50n 5 51n 5
             60n 5 61n 5 70n 5 71n 0 80n 0 81n 5 90n 5)

* Main Circuit
Ma 1 a o 1 CMOSP W=6U L=2U
Mb o a 2 0 CMOSN W=3U L=2U
Mc 1 b o 1 CMOSP W=6U L=2U
Md 2 b 3 0 CMOSN W=3U L=2U
Me 1 c o 1 CMOSP W=6U L=2U
Mf 3 c 4 0 CMOSN W=3U L=2U
Mg 1 d o 1 CMOSP W=6U L=2U
Mh 4 d 0 0 CMOSN W=3U L=2U

C1 o 0 .1P

* Simulation Parameters
.TRAN .0001N 90N

.END
```

Figure 165. Four Input NAND gate SPICE model source code.

5. Three Input AND

```
* File: and3.cir
* CMOS 3-input AND Transient Characteristics

* CMOSP & CMOSN model definitions
.INCLUDE cmos.cir
.INCLUDE subckt.cir

* Power Supplies
VDS 1 0 5

* Input Signals
VINa a 0 PWL(0n 5 10n 5 11n 5 20n 5 21n 5 30n 5 31n 5 40n 5 41n 5 50n 5 51n 0
             60n 0 61n 5 70n 5)
VINb b 0 PWL(0n 5 10n 5 11n 5 20n 5 21n 5 30n 5 31n 0 40n 0 41n 5 50n 5 51n 5
             60n 5 61n 5 70n 5)
VINc c 0 PWL(0n 5 10n 5 11n 0 20n 0 21n 5 30n 5 31n 5 40n 5 41n 5 50n 5 51n 5
             60n 5 61n 5 70n 5)

* Main Circuit
Xla a b c 2 1 0 NAND3
Xia 2 0 1 0 INV

C1 o 0 .1P

* Simulation Parameters
.TRAN .0001N 140N

.END
```

Figure 166. Three Input AND gate SPICE model source code.

6. Four Input AND

```
* File: and4.cir
* CMOS 4-input AND Transient Characteristics

* CMOSP & CMOSN model definitions
.INCLUDE cmos.cir

* Power Supplies
VDS 1 0 5

* Input Signals
VINa a 0 PWL(0n 5 10n 5 11n 5 20n 5 21n 5 30n 5 31n 5 40n 5 41n 5 50n 5 51n 0
             60n 0 61n 5 70n 5 71n 5 80n 5 81n 5 90n 5)
VINb b 0 PWL(0n 5 10n 5 11n 5 20n 5 21n 5 30n 5 31n 0 40n 0 41n 5 50n 5 51n 5
             60n 5 61n 5 70n 5 71n 5 80n 5 81n 5 90n 5)
VINc c 0 PWL(0n 5 10n 5 11n 0 20n 0 21n 5 30n 5 31n 5 40n 5 41n 5 50n 5 51n 5
             60n 5 61n 5 70n 5 71n 5 80n 5 81n 5 90n 5)
VINd d 0 PWL(0n 5 10n 5 11n 5 20n 5 21n 5 30n 5 31n 5 40n 5 41n 5 50n 5 51n 5
             60n 5 61n 5 70n 5 71n 0 80n 0 81n 5 90n 5)

* Main Circuit
Ma 1 a 5 1 CMOSP W=6U L=2U
Mb o a 2 0 CMOSN W=3U L=2U
Mc 1 b 5 1 CMOSP W=6U L=2U
Md 2 b 3 0 CMOSN W=3U L=2U
Me 1 c 5 1 CMOSP W=6U L=2U
Mf 3 c 4 0 CMOSN W=3U L=2U
Mg 1 d 5 1 CMOSP W=6U L=2U
Mh 4 d 0 0 CMOSN W=3U L=2U
Xia 5 0 1 0 INV

C1 o 0 .1P

* Simulation Parameters
.TRAN .0001N 90N

.END
```

Figure 167. Four Input AND gate SPICE model source code.

7. Two Input NOR

```
* File: nor2.cir
* CMOS 2-input NOR Transient Characteristics

* CMOSP & CMOSN model definitions
.INCLUDE cmos.cir
.INCLUDE subckt.cir

* Power Supplies
VDS 1 0 5

* Input Signals
VINa a 0 PWL(0n 0 10n 0 11n 0 20n 0 21n 0 30n 0 31n 5 40n 5 41n 0 50n 0 51n 5
             60n 5 61n 0 70n 0 71n 5 80n 5 81n 5 90n 5 91n 0 100n 0 101n 5
             110n 5 111n 5 120n 5 121n 0 130n 0)
VINb b 0 PWL(0n 0 10n 0 11n 5 20n 5 21n 0 30n 0 31n 0 40n 0 41n 0 50n 0 51n 5
             60n 5 61n 5 70n 5 71n 5 80n 5 81n 0 90n 0 91n 5 100n 5 101n 0
             110n 0 111n 5 120n 5 121n 0 130n 0)

* Main Circuit
Xla a b o 1 0 NOR2

C1 o 0 .1P

* Simulation Parameters
.TRAN .0001N 130N

.END
```

Figure 168. Two Input NOR gate SPICE model source code.

8. Three Input NOR

```
* File: nor3.cir
* CMOS 3-input NOR Transient Characteristics

* CMOSP & CMOSN model definitions
.INCLUDE cmos.cir
.INCLUDE subckt.cir

* Power Supplies
VDS 1 0 5

* Input Signals
VINa a 0 PWL(0n 0 10n 0 11n 0 20n 0 21n 0 30n 0 31n 0 40n 0 41n 0 50n 0 51n 5
             60n 5 61n 0 70n)
VINb b 0 PWL(0n 0 10n 0 11n 0 20n 0 21n 0 30n 0 31n 5 40n 5 41n 0 50n 0 51n 0
             60n 0 61n 0 70n)
VINc c 0 PWL(0n 0 10n 0 11n 5 20n 5 21n 0 30n 0 31n 0 40n 0 41n 0 50n 0 51n 0
             60n 0 61n 0 70n)

* Main Circuit
Xla a b c o 1 0 NOR3

C1 o 0 .1P

* Simulation Parameters
.TRAN .0001N 70N

.END
```

Figure 169. Three Input NOR gate SPICE model source code.

9. Two Input XOR

```
* File: xor2.cir
* CMOS 2-input XOR Transient Characteristics

* CMOSP & CMOSN model definitions
.INCLUDE cmos.cir
.INCLUDE subckt.cir

* Power Supplies
VDS 1 0 5

* Input Signals
VINa a 0 PWL(0n 0 10n 0 11n 0 20n 0 21n 0 30n 0 31n 5 40n 5 41n 0 50n 0 51n 5
             60n 5 61n 0 70n 0 71n 5 80n 5 81n 5 90n 5 91n 0 100n 0 101n 5
             110n 5 111n 5 120n 5 121n 0 130n 0)
VINb b 0 PWL(0n 0 10n 0 11n 5 20n 5 21n 0 30n 0 31n 0 40n 0 41n 0 50n 0 51n 5
             60n 5 61n 5 70n 5 71n 5 80n 5 81n 0 90n 0 91n 5 100n 5 101n 0
             110n 0 111n 5 120n 5 121n 0 130n 0)

* Main Circuit
Xla a b o 1 0 XOR2

C1 o 0 .1P

* Simulation Parameters
.TRAN .0001N 130N

.END
```

Figure 170. Two Input XOR gate SPICE model source code.

10. Two Input XNOR

```
* File: xnor2g.cir
* CMOS 2-input XNOR-gate Transient Characteristics

* CMOSP & CMOSN model definitions
.INCLUDE cmos.cir

* Power Supplies
VDS 1 0 5

* Input Signals
VINa a 0 PULSE(5 0 5.5N 1N 1N  9N 20N)
VINb b 0 PULSE(5 0 .5N 1N 1N 24N 50N)

* Main Circuit
Ma 1 a 2 1 CMOSP W=6U L=2U
Mb 2 a 0 0 CMOSN W=3U L=2U
Mc 1 2 4 1 CMOSP W=6U L=2U
Md 4 2 0 0 CMOSN W=3U L=2U
Me 4 3 o 1 CMOSP W=6U L=2U
Mf 4 b o 0 CMOSN W=3U L=2U
Mg 2 b o 1 CMOSP W=6U L=2U
Mh 2 3 o 0 CMOSN W=3U L=2U
Mi 1 b 3 1 CMOSP W=6U L=2U
Mj 3 b 0 0 CMOSN W=3U L=2U

C1 o 0 .1P

* Simulation Parameters
.TRAN .005N 100N

.END
```

Figure 171. Two Input XNOR gate SPICE model source code.

11. D Flip Flop with Clear

```
* File: dflopgc.cir
* CMOS D-FLIP/FLOP gated w/ Clear Transient Characteristics

* CMOSP & CMOSN model definitions
.INCLUDE cmos.cir
.INCLUDE subckt.cir

* Power Supplies
VDS 1 0 5

* Input Signals
VINd d 0 PULSE(0 5 .5N 1N 1N 19N 40N)
VINnc nc 0 PWL(0n 5 48n 5 49n 0 50n 0 51n 5 112n 5 113n 0 117n 0 118n 5
               125n 5)
VINclk clk 0 PULSE(0 5 2.5N 1N 1N 9N 20N)

* Main Circuit
Xda d clk nc q nq 1 0 DFLOPGC

C1 q 0 .1P
C2 nq 0 .1P

* Simulation Parameters
.TRAN .001N 125N

.END
```

Figure 172. D Flip Flop with Clear logic element SPICE model source code.

12. Two Input Multiplexer

```
* File: mux2.cir
* CMOS 2-input MUX Transient Characteristics

* CMOSP & CMOSN model definitions
.INCLUDE cmos.cir
.INCLUDE subckt.cir

* Power Supplies
VDS 1 0 5

* Input Signals
VINa a 0 PWL(0n 5 10n 5 11n 5 20n 5 21n 5 30n 5 31n 5 40n 5 41n 5 50n 5 51n 0
             60n 0 61n 5 70n 5)
VINb b 0 PWL(0n 5 10n 5 11n 5 20n 5 21n 5 30n 5 31n 0 40n 0 41n 5 50n 5 51n 5
             60n 5 61n 5 70n 5)
VINs s 0 PWL(0n 5 10n 5 11n 0 20n 0 21n 5 30n 5 31n 5 40n 5 41n 5 50n 5 51n 5
             60n 5 61n 5 70n 5)

* Main Circuit
Xla a b s o 1 0 MUX

C1 o 0 .1P

* Simulation Parameters
.TRAN .0001N 140N

.END
```

Figure 173. Two Input MUX logic element SPICE model source code.

C. VLSI LAYOUT

1. Legend of Layout Layers

Graphic Symbol	Layer Description
	N well - a region of the silicon substrate that has more free electrons than free holes.
	P well - a region of the silicon substrate that has more free holes than free electrons.
	Active - layout area to be implanted with impurities to provide primary charge carriers.
	Active X - connection shaft that allows contact between metal 1 and the active area.
	P select - boundary of area to be implanted with an impurity providing free holes.
	N select - boundary of area to be implanted with an impurity providing free electrons.
	Poly 1 - polysilicon doped for improved conduction; primarily used for FET gates.
	Poly 1 Connect - connection shaft that allows contact between metal 1 and polysilicon.
	Metal 1 - lowest layer of aluminum used to route signals and power.
	Metal 2 - upper layer of aluminum used to route signals and power.
	Via X - connection shaft between metal 1 layer and metal 2.

Table 25. Legend for Layers used in VLSI Layout.

2. Inverter

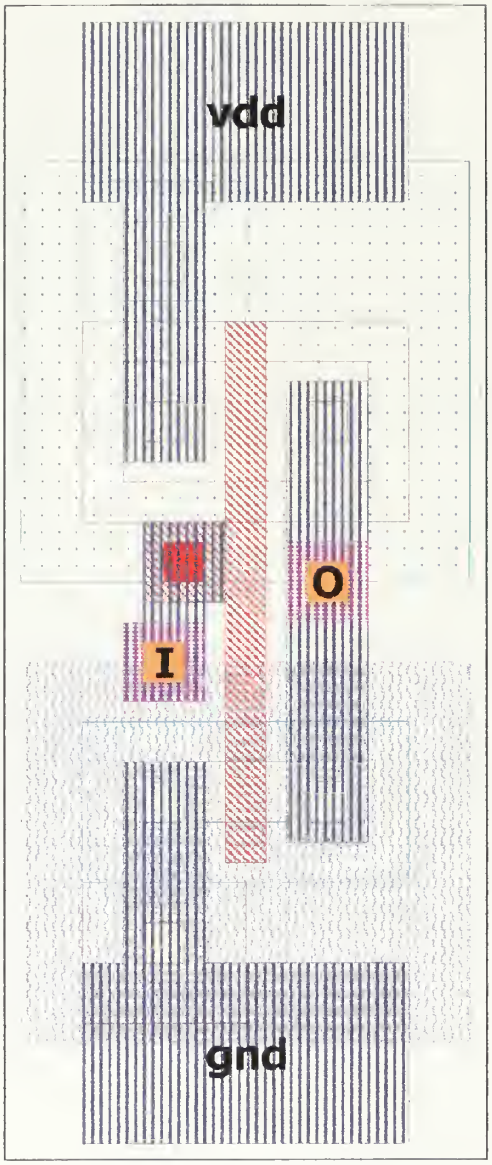


Figure 174. Inverter layout.

3. Two Input NAND

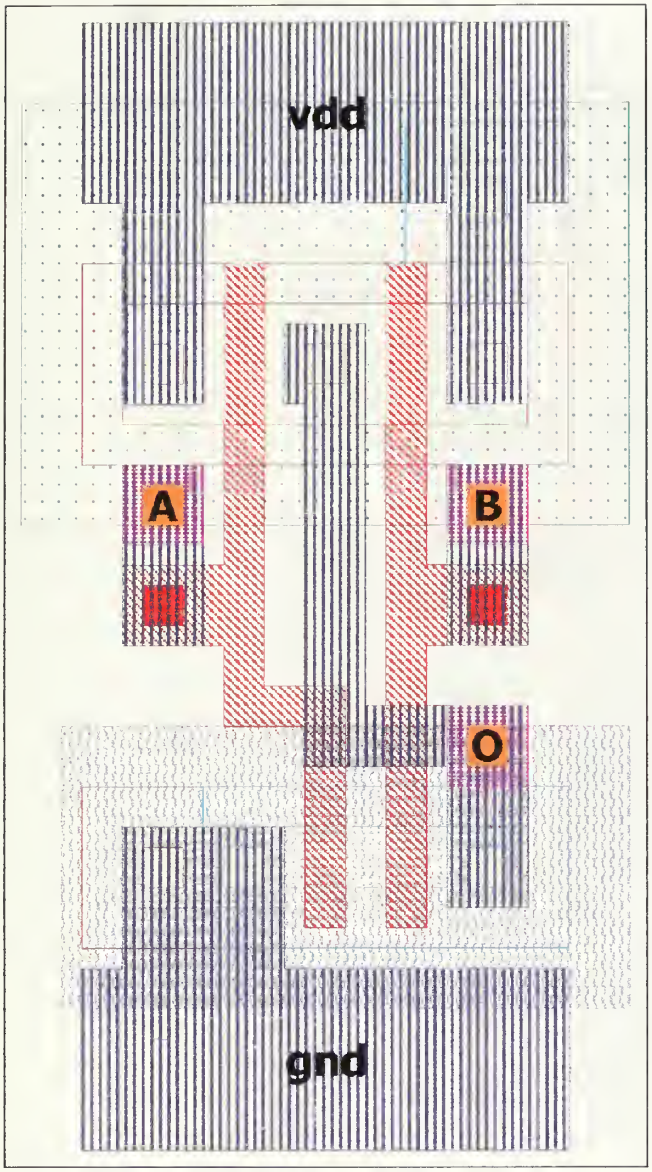


Figure 175. Two Input NAND Gate layout.

4. Three Input NAND

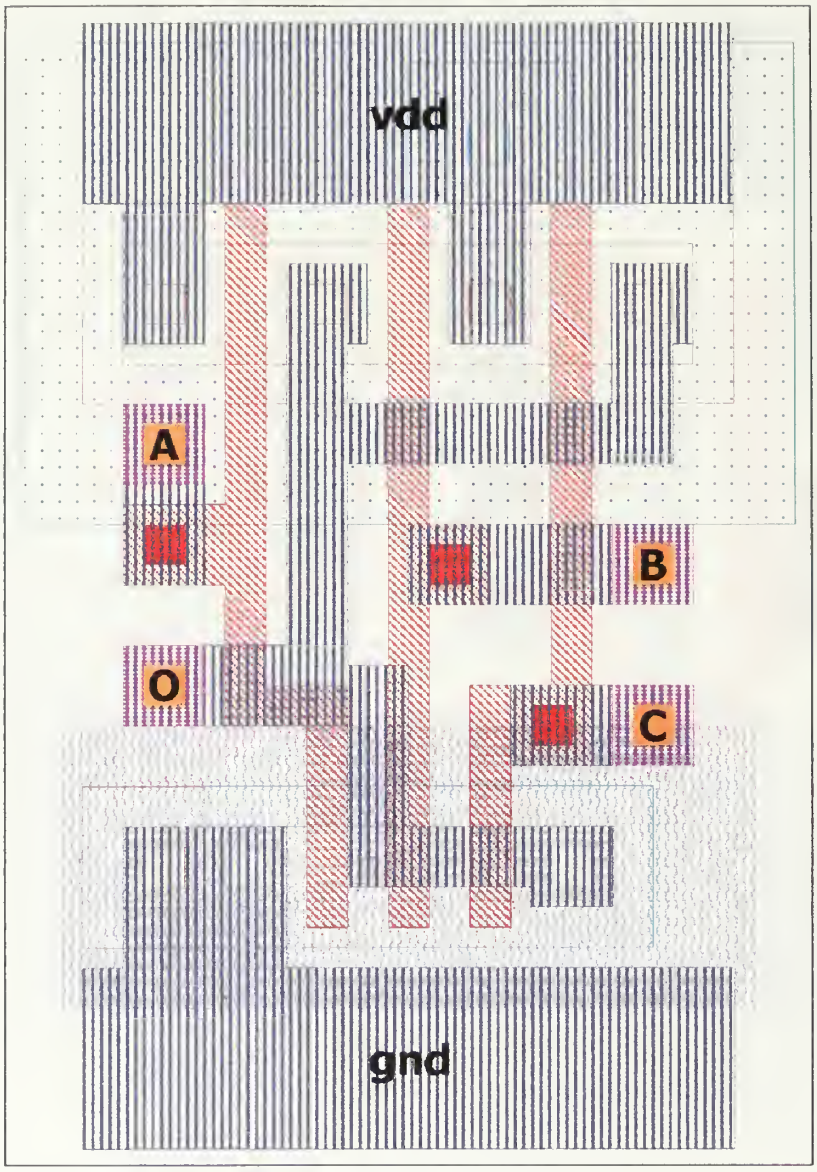


Figure 176. Three Input NAND Gate layout.

5. Four Input NAND

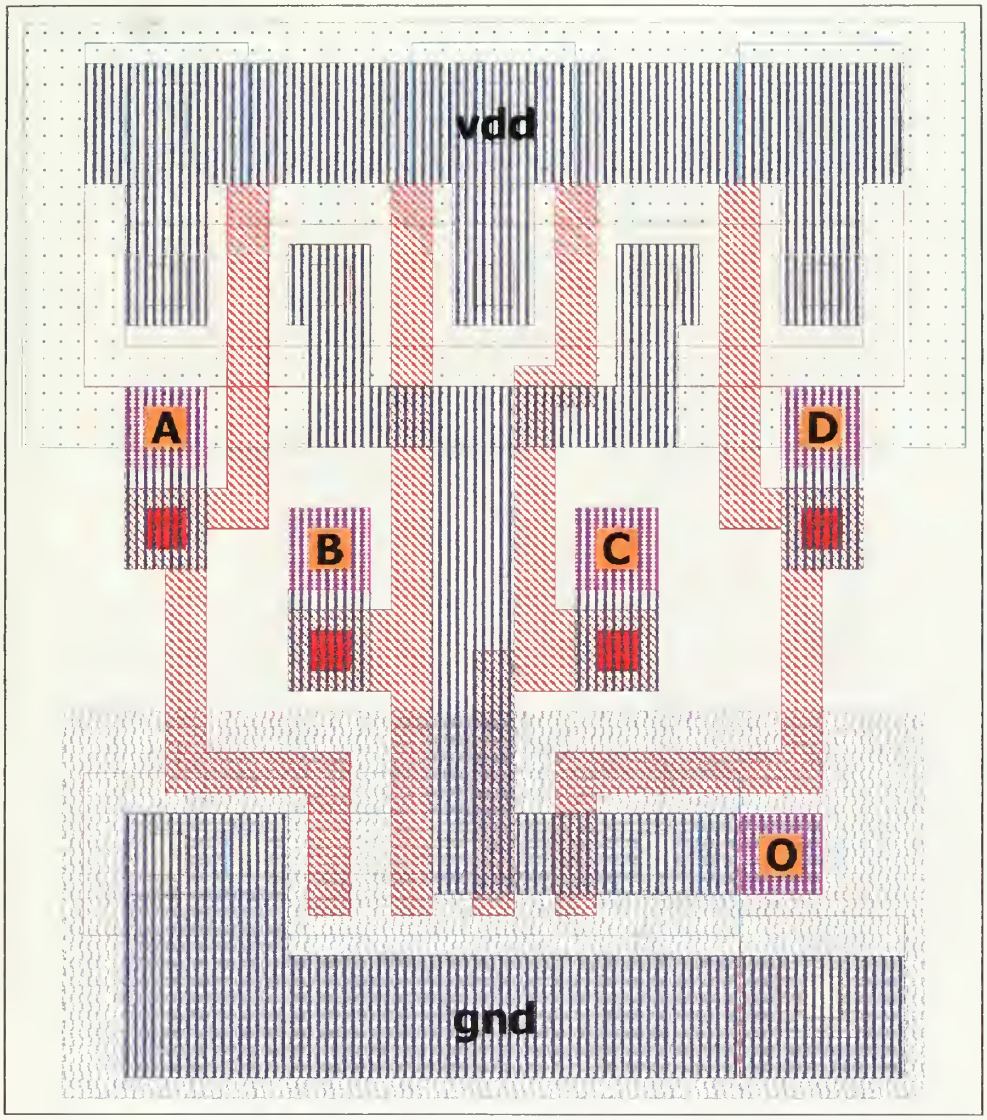


Figure 177. Four Input NAND Gate layout.

6. Three Input AND

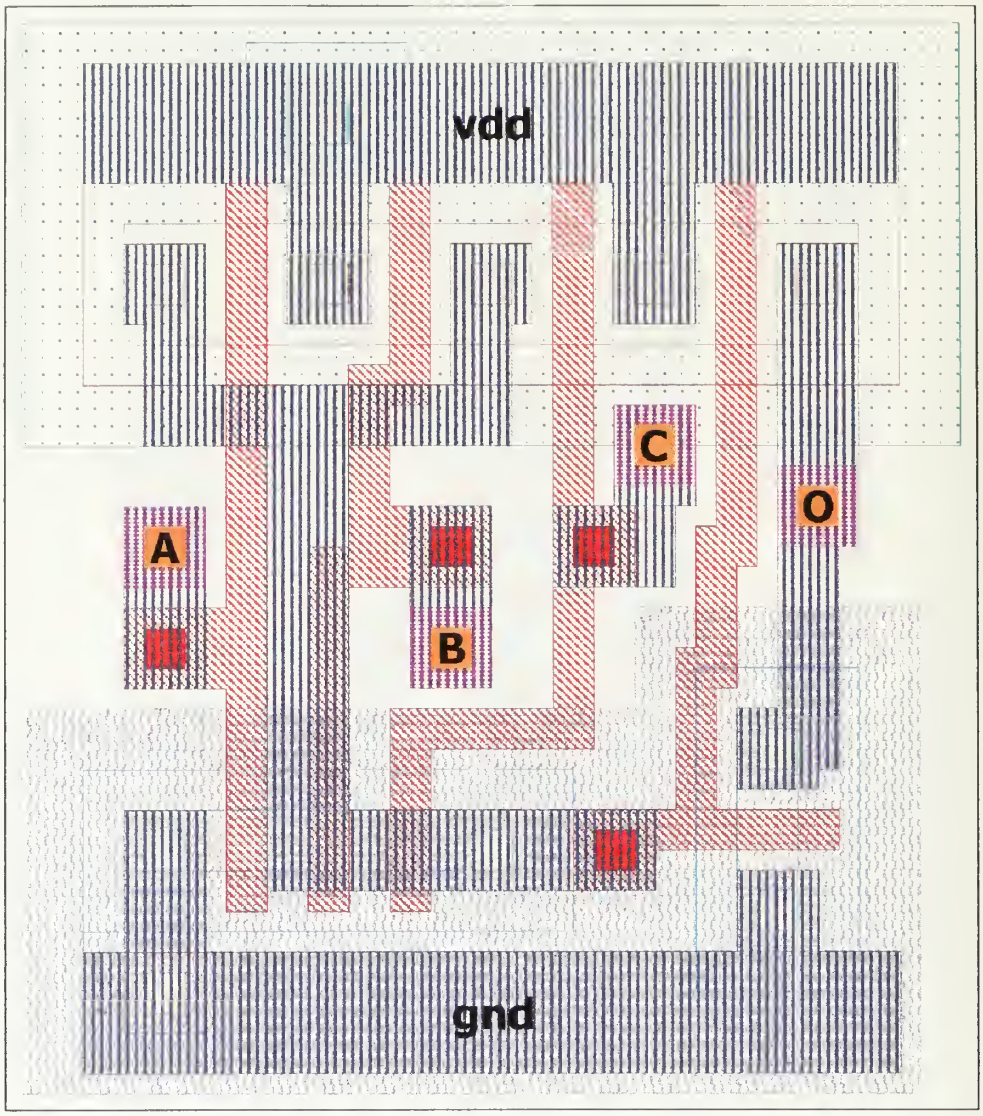


Figure 178. Three Input AND Gate layout.

7. Four Input AND

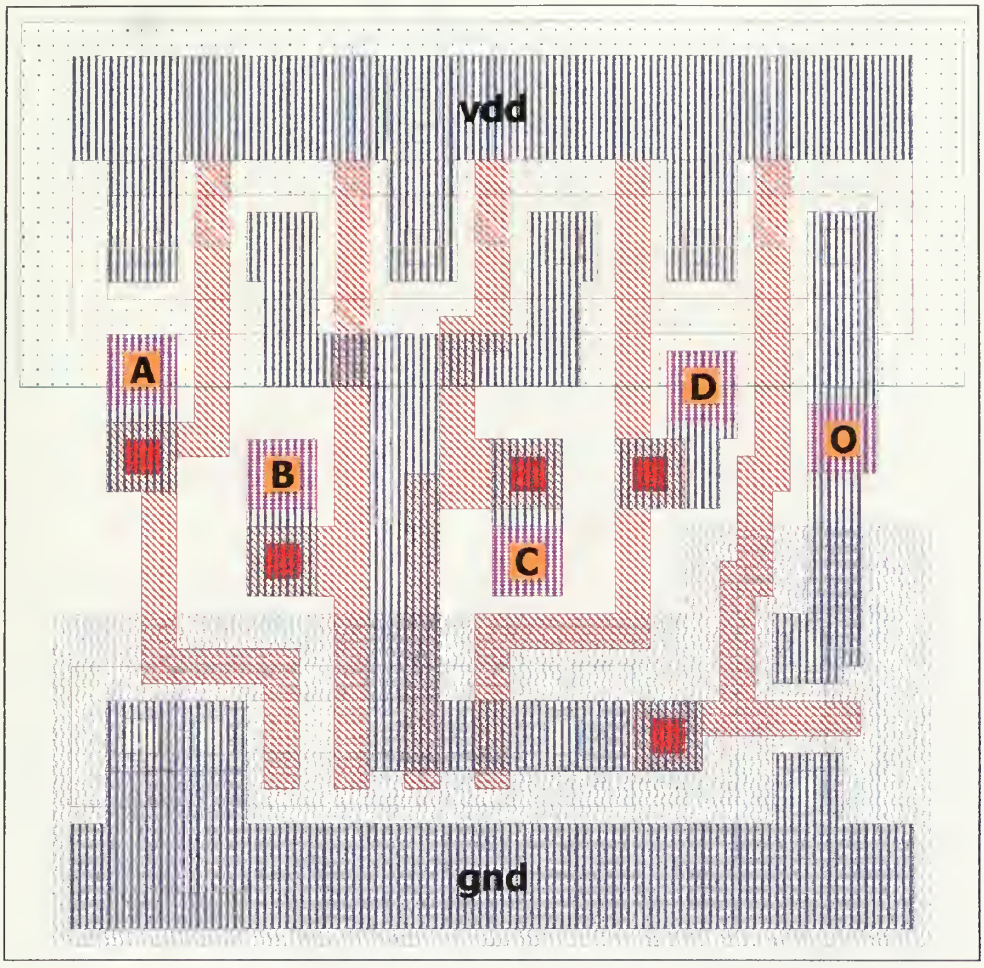


Figure 179. Four Input AND Gate layout.

8. Two Input NOR

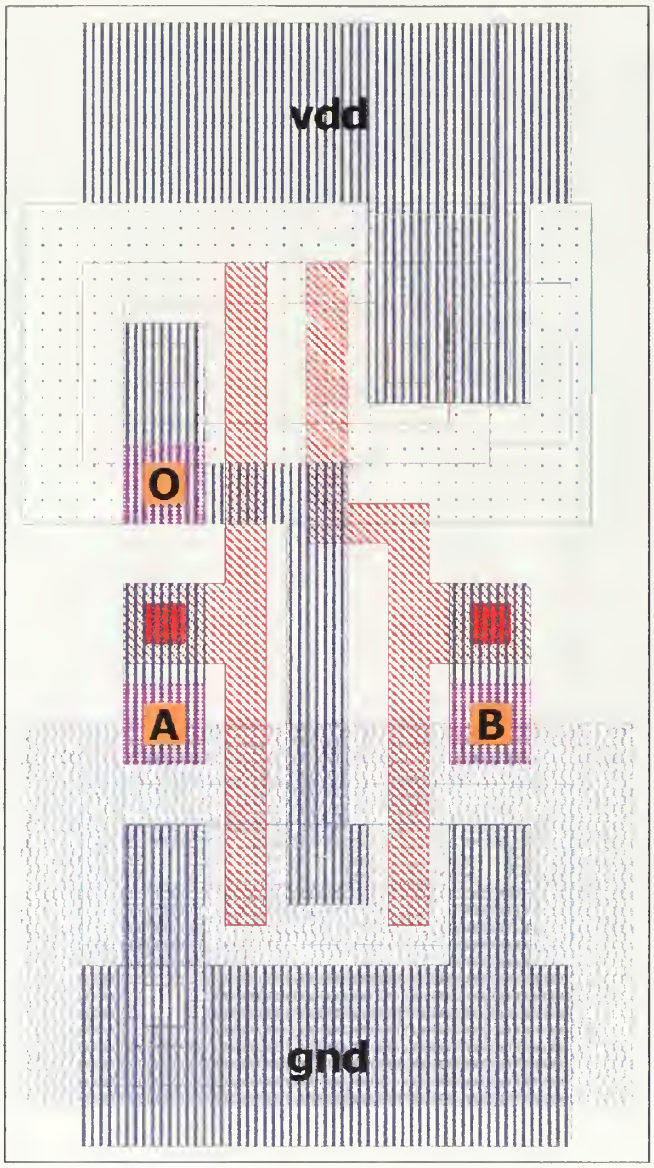


Figure 180. Two Input NOR Gate layout.

9. Three Input NOR

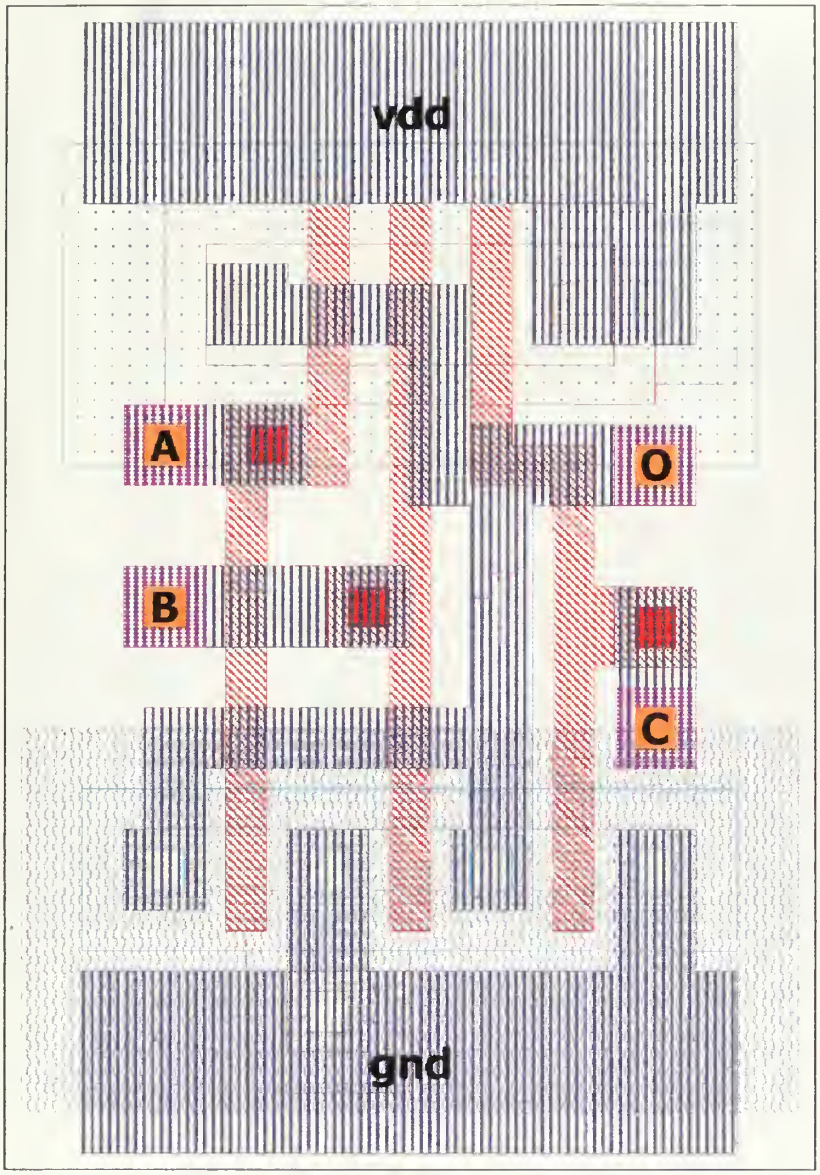


Figure 181. Three Input NOR Gate layout.

10. Two Input XOR

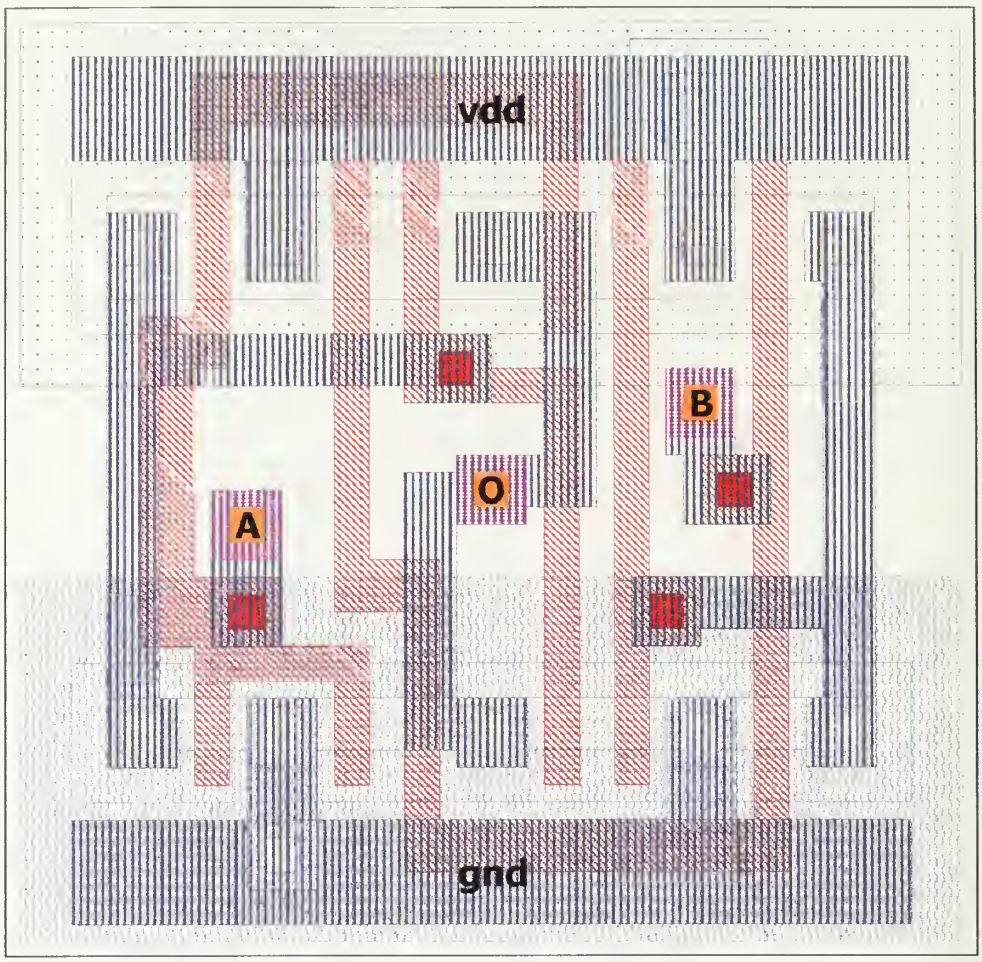


Figure 182. Two Input XOR Gate layout.

11. Two Input XNOR

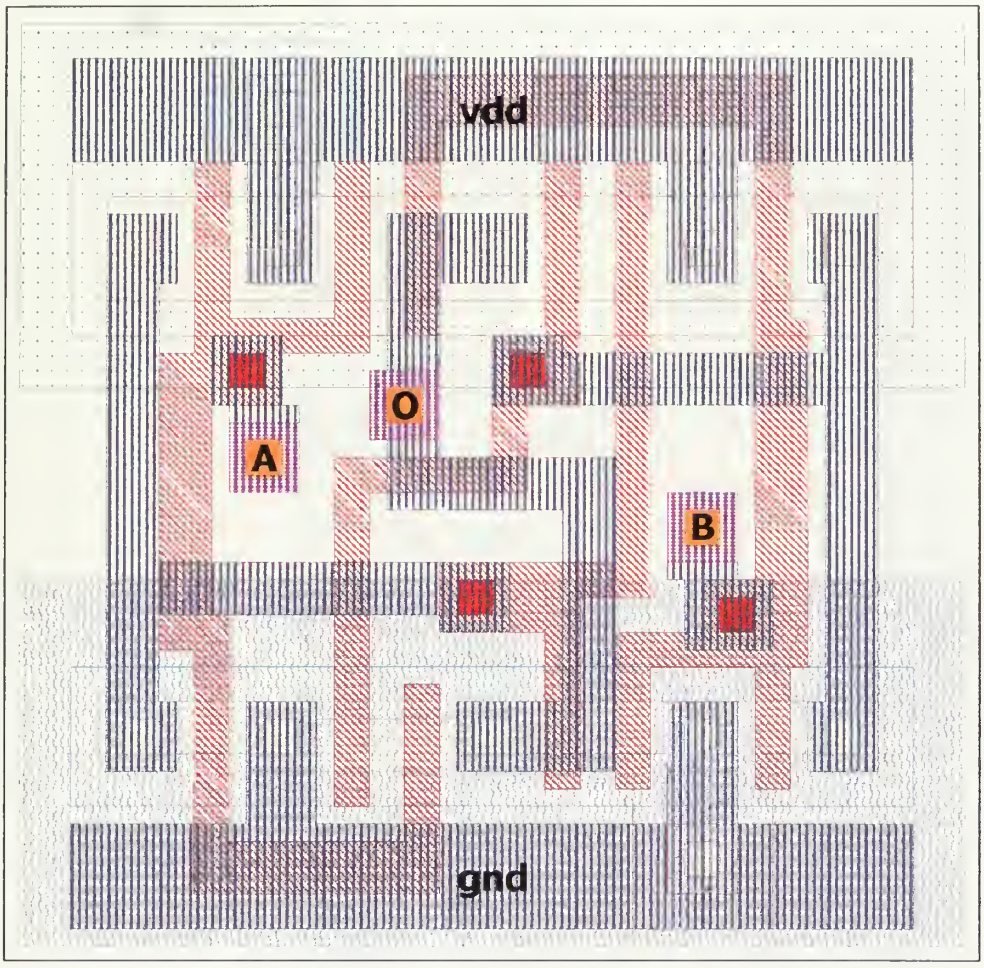


Figure 183. Two Input XNOR Gate layout.

12. D Flip Flop with Clear

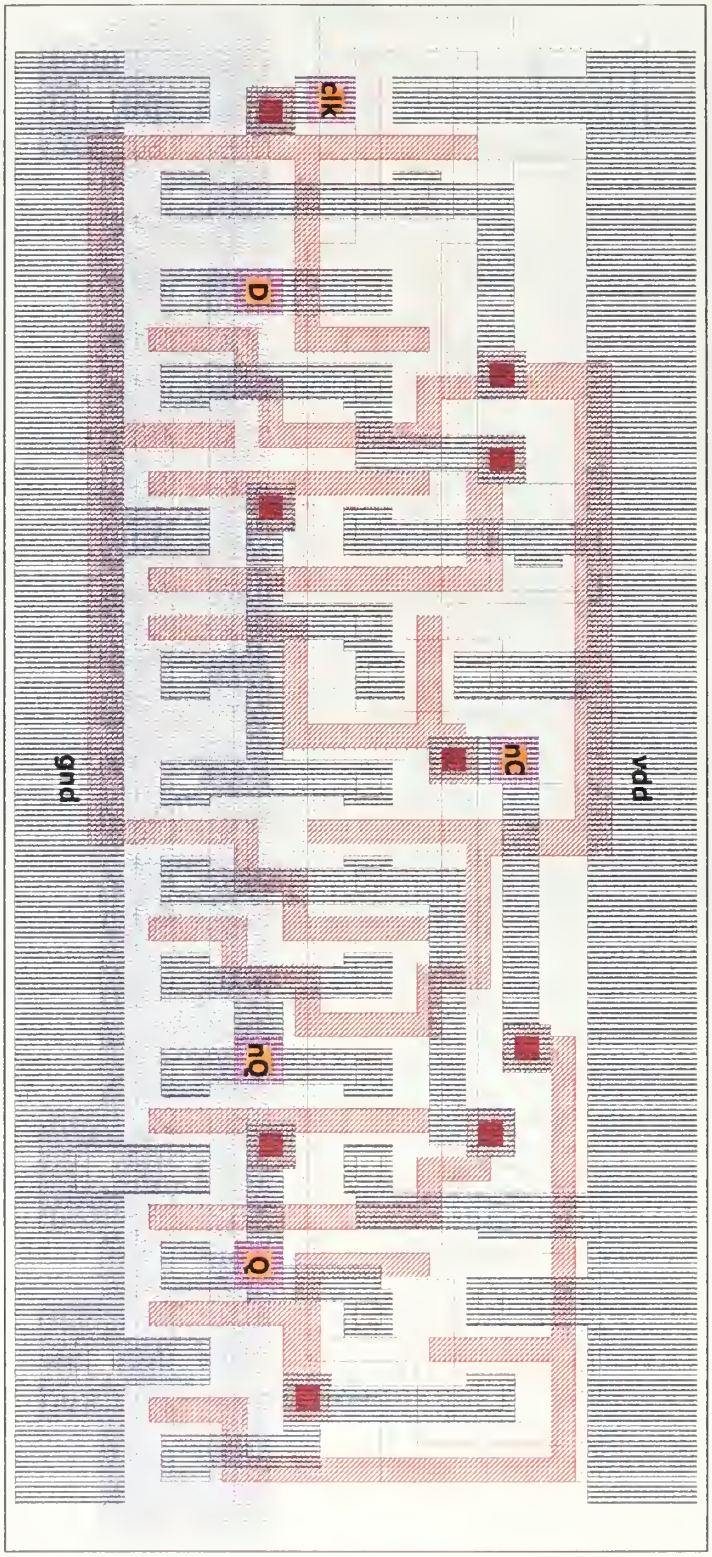


Figure 184. D Flip Flop with Clear layout.

13. Two Input Multiplexer

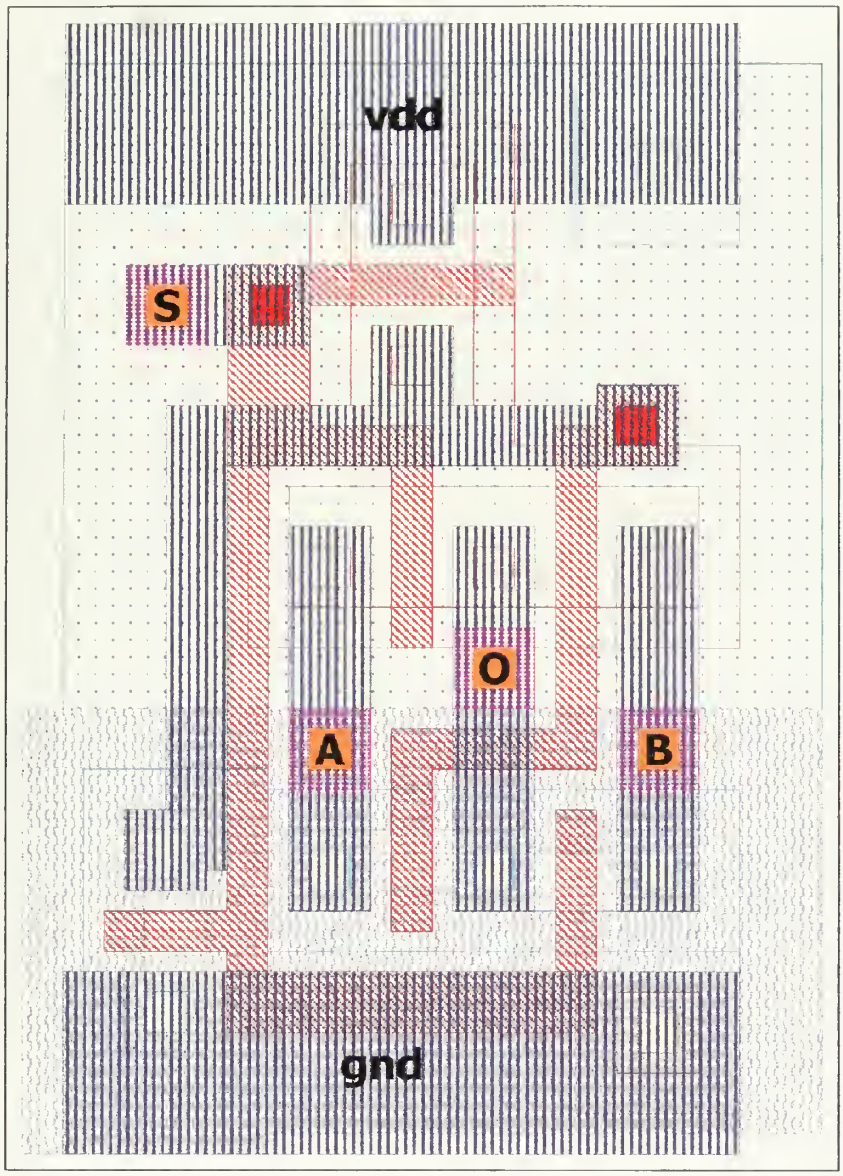


Figure 185. Two Input Multiplexer layout.

APPENDIX F. PARALLEL DATA MODULATOR DESIGN

Each module used to create the Parallel Port Data Modulator was first modeled using Verilog®. When proper system operation was obtained, ABEL™ was used to create the required JEDEC format data files for PLD programming. Reference 5 contains extensive information regarding PLD programming using ABEL™ and includes an educational version of the ABEL™ software written by Data I/O Corporation.

A. COMMAND MODULATOR DESIGN USING VERILOG®

The Parallel Port Data Modulator was modeled and tested using Verilog®. The source code for the test program is provided in Figure 186.

```
//*********************************************************************
// File:  xmit_test.v
//
// Description:  Test bench for Parallel Port Data Modulator
//
// Author: Jeff Link
//*********************************************************************

module xmit_test;

  reg  [7:0] d;
  reg  Str;
  wire [7:0] q;
  wire [3:0] s;
  reg Vdd,Gnd;

  clock    clk1 (clk);
  reg4_pls rg4a (d[7:4],Vdd,Vdd,clk,q[7:4],Ea,Oa);
  reg4_pls rg4b (d[3:0],Ea,Oa,clk,q[3:0],eq,odd);
  mux8t1   mx   (q,s[2:0],mxout);
  control  cntl (Str,eq,mxout,odd,clk,s,ackn,busy,out);


  initial  begin
    Vdd=1;
    Gnd=0;
    d=55;
    Str=0;
    $display("\t\t\t   d     out  ackn busy  s");
    $monitor("time %0d  \t%b  %b    %b    %b   %d",$time,d,out,ackn,busy,s);
    #5;
    Str=1;
    #20 Str=0;
    #400

    d=85;
    #20
    Str=1;
    #20 Str=0;
    #400

    d=205;
    #20
    Str=1;
    #20 Str=0;
    #400
    $finish;
  end

/*  always @ (q) begin
    if (odd == ^q) begin
      $display("time %0d  \t%b %b  %b  %b  Parity Error",$time,d,q,eq,odd);
    end
  end
*/
endmodule
```

Figure 186. Test bench for Parallel Port Data Modulator Verilog® model source code.

1. Four-Bit Register with Equality and Parity Calculation

The four-bit register with equality and parity calculation was modeled and tested using Verilog®. The source code for the test program is provided in Figure 187 and the source code for the model is provided in Figure 188.

```
//*********************************************************************
// File:  reg4_pls_test.v
//
// Description:  Test bench for Four-Bit Register w/ Equal & Parity
//
// Author: Jeff Link
//*********************************************************************

module reg4_pls_test;

  reg  [7:0] d;
  wire clk;
  wire [7:0] q;
  reg Vdd,Gnd;

  clock  clk1 (clk);
  reg4_pls rg4a (d[7:4],Vdd,Vdd,clk,q[7:4],Ea,Oa);
  reg4_pls rg4b (d[3:0],Ea,Oa,clk,q[3:0],eq,odd);

  initial  begin
    Vdd=1;
    Gnd=0;
    d=0;
    $display("\t\t\t   d        q       eq odd");
    $monitor("time %0d  \t%b %b  %b  %b",$time,d,q,eq,odd);
    #5;
    for (d=0; d<255; d=d+1)  begin
      #40;
    end

    $finish;
  end

endmodule
```

Figure 187. Test bench for Four-Bit Register Verilog® model source code.

```
//*********************************************************************
// File:  reg4_pls.v
//
// Description:  Behavioral Model of Four-Bit Register w/ Equal & Parity.
//
// Author: Jeff Link
//*********************************************************************

module reg4_pls (d,Ein,Oin,clk,q,Eout,Oout);
  input d,Ein,Oin,clk;
  wire  [3:0] d;
  output q,Eout,Oout;
  reg [3:0] q;
  reg Eout,Oout;

  always @(d)  begin
    Eout = (q == d)&Ein;
  end

  always @(Ein)  begin
    Eout = (q == d)&Ein;
  end

  always @(Oin)  begin
    Oout = (^q)^Oin;
  end

  always @(posedge clk)  begin
    q = d;
    Eout = (q == d)&Ein;
    Oout = (^q)^Oin;
  end

endmodule
```

Figure 188. Four-Bit Register Verilog® model source code.

2. Control State Machine

The control state machine was modeled and tested using Verilog®. The source code for the test program is provided in Figure 189 and the source code for the model is provided in Figure 190.

```
//*********************************************************************
// File:  control_test.v
//
// Description:  Test bench for Control State Machine
//
// Author: Jeff Link
//*********************************************************************

module control_test;

  reg  Str,EqIn,Din,Par;
  wire clk;
  wire [3:0] s;
  reg Vdd,Gnd;

  clock   clk1 (clk);
  control cntl (Str,EqIn,Din,Par,clk,s,ackn,busy,out);

  initial  begin
    Vdd=1;
    Gnd=0;
    $display("\t\t\tStr EqIn Din Par s ackn busy out");
    $monitor("time %0d  \t %b   %b   %b   %b  %d  %b %b  %b",
                    $time,Str,EqIn,Din,Par,s,ackn,busy,out);
    Str=0;
    EqIn=0;
    Din=1;
    Par=0;
    #5;
    Str=1;
    #20;
    EqIn=1;
    #20;
    Str=0;
    #20;
    Din=0;
    #20;
    Din=1;
    #20;
    Din=1;
    #20;
    Din=0;
    #20;
    Din=1;
    #20;
    Din=0;
    #20;
    Din=1;
    #200;
    $finish;
  end

/*  always @ (q) begin
    if (odd == ^q) begin
      $display("time %0d  \t%b %b  %b  %b  Parity Error",$time,d,q,eq,odd);
    end
  end
*/
endmodule
```

Figure 189. Test bench for Control State Machine Verilog® model source code.

```
//*****************************************************************************
// File:  control.v
//
// Description:  Behavioral Model of Control State Machine
//
// Author: Jeff Link
//*****************************************************************************

module control (Str,EqIn,Din,Par,clk,st,ackn,busy,out);
  input Str,EqIn,Din,Par,clk;
  output st,ackn,busy,out;
  reg [3:0] st;
  reg ackn,busy,out;

  initial begin
    st = 0;
    ackn=0;
    busy=0;
    out=1;
  end

  always @(posedge clk)  begin
    case (st)
       0: if (Str&EqIn) st = 4;
       4: st = 12;
      12: st = 8;
       8: st = 9;
       9: st = 11;
      11: st = 10;
      10: st = 14;
      14: st = 15;
      15: st = 13;
      13: st = 5;
       5: st = 1;
       1: st = 0;
    endcase
  end

  always @(st)  begin
    if (st == 0) busy = 0;
    else busy = 1;
    if (st == 4) ackn = 1;
    else ackn = 0;
    if (st == 0) out = 1;
    else if (st == 1) out = 1;
    else if (st == 4) out = 0;
    else if (st == 5) out = Par;
    else out = Din;
  end

  always @(Din)  begin
    if (st == 0) out = 1;
    else if (st == 1) out = 1;
    else if (st == 4) out = 0;
    else if (st == 5) out = Par;
    else out = Din;
  end

endmodule
```

Figure 190. Control State Machine Verilog® model source code.

3. Eight-to-One Multiplexer

The eight-to-one multiplexer was modeled and tested using Verilog®. The source code for the test program is provided in Figure 191 and the source code for the model is provided in Figure 192.

```
//*********************************************************************
// File:  mux8t1_test.v
//
// Description:  Test bench for Eight to One Multiplexer
//
// Author: Jeff Link
//*********************************************************************

module mux8t1_test;

  reg  [7:0] d;
  reg  [2:0] sel;
  reg Vdd,Gnd;

  mux8t1 mx1 (d,sel,out);

  initial  begin
    Vdd=1;
    Gnd=0;
    d=0;
    $display("\t\t\t   d       sel   out");
    $monitor("time %0d  \t%b %b  %b",$time,d,sel,out);
    #5;
    for (d=55; d<199; d=d+13)  begin
      for (sel=0; sel<7; sel=sel+1)  begin
        #40;
      end
      #40;
    end

    $finish;
  end

/*  always @ (q) begin
    if (odd == ^q) begin
      $display("time %0d  \t%b %b  %b  %b  Parity Error",$time,d,q,eq,odd);
    end
  end
*/
endmodule
```

Figure 191. Test bench for Eight-to-One Multiplexer Verilog® model source code.

```
//*****************************************************************************
// File:  mux8t1.v
//
// Description:  Behavioral Model of Eight to One Multiplexer.
//
// Author: Jeff Link
//*****************************************************************************

module mux8t1 (a,sel,out);
  input a,sel;
  wire  [7:0] a;
  wire  [2:0] sel;
  output out;
  reg out;

  always @(sel)  begin
    case (sel)
      0:  out = a[6];
      1:  out = a[5];
      2:  out = a[3];
      3:  out = a[4];
      4:  out = a[7];
      5:  out = a[0];
      6:  out = a[2];
      7:  out = a[1];
    endcase
  end

endmodule
```

Figure 192. Eight-to-One Multiplexer Verilog® model source code.

B. COMMAND MODULATOR IMPLEMENTATION USING ABEL™

When operating properly, the Parallel Port Data Modulator elements defined using Verilog® were converted to JEDEC file format using ABEL™. The source code created for this conversion is included in the following subsections. These programs were compiled and optimized to create the JEDEC format data files needed for PLD programming.

1. Four-Bit Register with Equality and Parity Calculation

The four-bit register with equality and parity calculation was converted from Verilog® source code to JEDEC format using the ABEL™ code provided in Figure 193.

```
Module reg4_pls
Title 'Four-Bit Register with Equality & Parity outputs'
    Clk      pin 1;
    D4..D1   pin 2..5;
    Q4..Q1   pin 19..16 istype 'reg,buffer';
    !Ein     pin 8;
    Oin      pin 9;
    ODD      pin 14 istype 'com';
    EQ       pin 15 istype 'com';
    !Eout    pin 13 istype 'com';
    Oout     pin 12 istype 'com';
    Input  = [D4..D1];
    Output = [Q4..Q1];

Equations
    Output     := Input;
    Output.clk = !Clk;
    EQ          = (Output == Input);
    Eout        = EQ & Ein;
    ODD         = Q4 $ Q3 $ Q2 $ Q1;
    Oout        = ODD $ Oin;

Test_Vectors ([Clk,Input,!Ein,Oin] -> [Output,!Eout,Oout])
              [ 0 , ^h0 ,  0 , 0 ] -> [ ^h0  ,  0  , 0  ];
              [ 0 , ^h0 ,  0 , 1 ] -> [ ^h0  ,  0  , 1  ];
              [ 1 , ^h0 ,  1 , 1 ] -> [ ^h0  ,  1  , 1  ];
              [ 1 , ^h0 ,  1 , 0 ] -> [ ^h0  ,  1  , 0  ];
              [ 0 , ^h0 ,  0 , 0 ] -> [ ^h0  ,  0  , 0  ];
              [ 0 , ^h7 ,  0 , 1 ] -> [ ^h0  ,  1  , 1  ];
              [ 1 , ^h7 ,  0 , 0 ] -> [ ^h0  ,  1  , 0  ];
              [ 1 , ^h7 ,  0 , 0 ] -> [ ^h0  ,  1  , 0  ];
              [ 0 , ^h7 ,  0 , 0 ] -> [ ^h7  ,  0  , 1  ];
              [ 0 , ^h7 ,  0 , 1 ] -> [ ^h7  ,  0  , 0  ];
              [ 1 , ^h7 ,  0 , 0 ] -> [ ^h7  ,  0  , 1  ];
              [ 1 , ^hF ,  0 , 0 ] -> [ ^h7  ,  1  , 1  ];
              [ 0 , ^hF ,  0 , 0 ] -> [ ^hF  ,  0  , 0  ];
              [ 0 , ^hF ,  0 , 0 ] -> [ ^hF  ,  0  , 0  ];
              [ 1 , ^hF ,  1 , 0 ] -> [ ^hF  ,  1  , 0  ];
              [ 1 , ^hF ,  1 , 1 ] -> [ ^hF  ,  1  , 1  ];
              [ 0 , ^hF ,  0 , 1 ] -> [ ^hF  ,  0  , 1  ];
              [ 0 , ^h0 ,  0 , 0 ] -> [ ^hF  ,  1  , 0  ];
              [ 1 , ^h0 ,  1 , 0 ] -> [ ^hF  ,  1  , 0  ];
              [ 1 , ^h0 ,  0 , 1 ] -> [ ^hF  ,  1  , 1  ];
              [ 0 , ^h0 ,  0 , 1 ] -> [ ^h0  ,  0  , 1  ];
              [ 0 , ^h0 ,  0 , 0 ] -> [ ^h0  ,  0  , 0  ];
              [ 1 , ^h0 ,  0 , 0 ] -> [ ^h0  ,  0  , 0  ];
              [ 1 , ^h2 ,  0 , 0 ] -> [ ^h0  ,  1  , 0  ];
              [ 0 , ^h2 ,  0 , 0 ] -> [ ^h2  ,  0  , 1  ];
              [ 0 , ^h2 ,  0 , 1 ] -> [ ^h2  ,  0  , 0  ];
              [ 1 , ^h2 ,  1 , 0 ] -> [ ^h2  ,  1  , 1  ];
              [ 1 , ^h2 ,  1 , 1 ] -> [ ^h2  ,  1  , 0  ];
              [ 0 , ^h2 ,  1 , 0 ] -> [ ^h2  ,  1  , 1  ];
              [ 0 , ^h2 ,  0 , 0 ] -> [ ^h2  ,  0  , 1  ];
End
```

Figure 193. Four-Bit Register ABEL™ source code.

2. Control State Machine

The control state machine was converted from Verilog® source code to JEDEC format using the ABEL™ code provided in Figure 194.

```
module control
title 'Control State Machine '

    Clk      pin 1;            "Inputs
    Din      pin 2;
    !Par     pin 9;
    Strb     pin 11;           "Strobe is active low
    !Ein     pin 8;
    s3..s0   pin 16..19 istype 'reg';  "State bits
    out      pin 14     istype 'reg,buffer';
    busy     pin 12     istype 'reg';
    Ackn     pin 13     istype 'reg';
    iStrb    pin 15     istype 'com';

Equations
    [s3..s0].clk   = Clk;
    out.clk        = Clk;
    busy.clk       = Clk;
    Ackn.clk       = Clk;
    iStrb          = !Strb;

State_Diagram [s3..s0]

   State 0:  out := 1;       "Idle state
             busy := 0;
             Ackn := 1;
             If (iStrb&Ein) Then 4 Else 0 ;

   State 4:  out := 1;       "Standby, wait for strobe to reset
             busy := 1;      "  so that you only send one byte.
             Ackn := 0;
             If (!iStrb) Then 5 Else 4 ;

   State 5:  out := 0;       "Start bit
             busy := 1;
             Ackn := 1;
             Goto 13;
```

Figure 194. Control State Machine ABEL™ source code.

```
State 13: out := Din;     "Data bits
          busy := 1;
          Ackn := 1;
          Goto 12;
State 12: out := Din;
          busy := 1;
          Ackn := 1;
          Goto 8;
State 8:  out := Din;
          busy := 1;
          Ackn := 1;
          Goto 10;
State 10: out := Din;
          busy := 1;
          Ackn := 1;
          Goto 14;
State 14: out := Din;
          busy := 1;
          Ackn := 1;
          Goto 15;
State 15: out := Din;
          busy := 1;
          Ackn := 1;
          Goto 11;
State 11: out := Din;
          busy := 1;
          Ackn := 1;
          Goto 9;
State 9:  out := Din;
          busy := 1;
          Ackn := 1;
          Goto 1;

State 1:  out := Par;     "Parity bit
          busy := 1;
          Ackn := 1;
          Goto 0;
```

Figure 194. Control State Machine ABEL™ source code. (continued)

```
Test_Vectors ([Clk,Din ,Strb,Ein,Par] -> [[s3..s0],out,busy,Ackn])
              [ 0 ,.X. , 1  , 0 ,.X.] -> [   0    ,.X.,.X. ,.X. ];
              [ 1 ,.X. , 1  , 0 ,.X.] -> [   0    , 1 , 0  , 1  ];
              [ 0 ,.X. , 0  , 0 ,.X.] -> [   0    , 1 , 0  , 1  ];
              [ 1 ,.X. , 0  , 0 ,.X.] -> [   0    , 1 , 0  , 1  ];
              [ 0 ,.X. , 0  , 1 ,.X.] -> [   0    , 1 , 0  , 1  ];
              [ 1 ,.X. , 0  , 1 ,.X.] -> [   4    , 1 , 0  , 1  ];
              [ 0 ,.X. , 0  , 1 ,.X.] -> [   4    , 1 , 0  , 1  ];
              [ 1 ,.X. , 0  , 1 ,.X.] -> [   4    , 1 , 1  , 0  ];
              [ 0 ,.X. , 0  , 1 ,.X.] -> [   4    , 1 , 1  , 0  ];
              [ 1 ,.X. , 0  , 1 ,.X.] -> [   4    , 1 , 1  , 0  ];
              [ 0 ,.X. , 0  , 1 ,.X.] -> [   4    , 1 , 1  , 0  ];
              [ 1 ,.X. , 1  , 1 ,.X.] -> [   4    , 1 , 1  , 0  ];
              [ 0 ,.X. , 1  , 1 ,.X.] -> [   4    , 1 , 1  , 0  ];
              [ 1 ,.X. , 1  , 1 ,.X.] -> [   5    , 1 , 1  , 0  ];
              [ 0 ,.X. , 1  , 1 ,.X.] -> [   5    , 1 , 1  , 0  ];
              [ 1 , 1  , 1  , 1 ,.X.] -> [  13    , 0 , 1  , 1  ];
              [ 0 , 1  , 1  , 1 ,.X.] -> [  13    , 0 , 1  , 1  ];
              [ 1 , 1  , 1  , 1 ,.X.] -> [  12    , 1 , 1  , 1  ];
              [ 0 , 0  , 1  , 1 ,.X.] -> [  12    , 1 , 1  , 1  ];
              [ 1 , 0  , 1  , 1 ,.X.] -> [   8    , 0 , 1  , 1  ];
              [ 0 , 0  , 1  , 1 ,.X.] -> [   8    , 0 , 1  , 1  ];
              [ 1 , 0  , 1  , 1 ,.X.] -> [  10    , 0 , 1  , 1  ];
              [ 0 , 1  , 1  , 1 ,.X.] -> [  10    , 0 , 1  , 1  ];
              [ 1 , 1  , 1  , 1 ,.X.] -> [  14    , 1 , 1  , 1  ];
              [ 0 , 0  , 1  , 1 ,.X.] -> [  14    , 1 , 1  , 1  ];
              [ 1 , 0  , 1  , 1 ,.X.] -> [  15    , 0 , 1  , 1  ];
              [ 0 , 1  , 1  , 1 ,.X.] -> [  15    , 0 , 1  , 1  ];
              [ 1 , 1  , 1  , 1 ,.X.] -> [  11    , 1 , 1  , 1  ];
              [ 0 , 0  , 1  , 1 ,.X.] -> [  11    , 1 , 1  , 1  ];
              [ 1 , 0  , 1  , 1 ,.X.] -> [   9    , 0 , 1  , 1  ];
              [ 0 , 1  , 1  , 1 ,.X.] -> [   9    , 0 , 1  , 1  ];
              [ 1 , 1  , 1  , 1 ,.X.] -> [   1    , 1 , 1  , 1  ];
              [ 0 ,.X. , 1  , 1 , 0 ] -> [   1    , 1 , 1  , 1  ];
              [ 1 ,.X. , 1  , 1 , 0 ] -> [   0    , 0 , 1  , 1  ];
              [ 0 ,.X. , 1  , 1 ,.X.] -> [   0    , 0 , 1  , 1  ];
              [ 1 ,.X. , 1  , 1 ,.X.] -> [   0    , 1 , 0  , 1  ];
              [ 0 ,.X. , 1  , 1 ,.X.] -> [   0    , 1 , 0  , 1  ];
              [ 1 ,.X. , 1  , 1 ,.X.] -> [   0    , 1 , 0  , 1  ];
              [ 0 ,.X. , 1  , 1 ,.X.] -> [   0    , 1 , 0  , 1  ];
              [ 1 ,.X. , 1  , 1 ,.X.] -> [   0    , 1 , 0  , 1  ];

End
```

Figure 194. Control State Machine ABEL™ source code. (continued)

3. Eight-to-One Multiplexer

The eight-to-one multiplexer was converted from Verilog® source code to JEDEC format using the ABEL™ code provided in Figure 195.

```
module mux8t1

Title 'Eight to One Multiplexer'
    Clk    pin 11;
    a7..a0 PIN 9..2;
    s2..s0 PIN 14..12;
    out    PIN 15 ISTYPE 'com';
    nClk   pin 16 istype 'com';

    A      = [a7..a0];
    Select = [s2..s0];

Equations
    out = (Select == 1) & a0
        # (Select == 3) & a1
        # (Select == 7) & a2
        # (Select == 6) & a3
        # (Select == 2) & a4
        # (Select == 0) & a5
        # (Select == 4) & a6
        # (Select == 5) & a7;
    nClk = !Clk;

Test_Vectors ([Select,  A ,Clk] -> [out,nClk])
              [  5   ,^hAA, 0 ] -> [ 1 , 1  ];
              [  4   ,^hAA, 1 ] -> [ 0 , 0  ];
              [  0   ,^hAA, 0 ] -> [ 1 , 1  ];
              [  2   ,^hAA, 0 ] -> [ 0 , 1  ];
              [  6   ,^hAA, 1 ] -> [ 1 , 0  ];
              [  7   ,^hAA, 1 ] -> [ 0 , 0  ];
              [  3   ,^hAA, 0 ] -> [ 1 , 1  ];
              [  1   ,^hAA, 1 ] -> [ 0 , 0  ];
              [  5   ,^h55, 0 ] -> [ 0 , 1  ];
              [  4   ,^h55, 1 ] -> [ 1 , 0  ];
              [  0   ,^h55, 0 ] -> [ 0 , 1  ];
              [  2   ,^h55, 0 ] -> [ 1 , 1  ];
              [  6   ,^h55, 1 ] -> [ 0 , 0  ];
              [  7   ,^h55, 1 ] -> [ 1 , 0  ];
              [  3   ,^h55, 0 ] -> [ 0 , 1  ];
              [  1   ,^h55, 1 ] -> [ 1 , 0  ];

End
```

Figure 195. Eight-to-One Multiplexer ABEL™ source code.

C. COMMAND MODULATOR ELEMENT INFORMATION

Compilation of the ABEL™ source code presented in the preceding section created the JEDEC format data files. These JEDEC data files were used to program the PLDs to perform the defined logic functions. Information is generated in the compilation process regarding utilization, performance, and layout. This chip information is written to a text file and the essential information from these files is included in the following subsections. Especially useful is the chip pin assignment diagram; required for laying out the printed circuit board.

1. Four-Bit Register with Equality and Parity Calculation

Information regarding the four-bit register with equality and parity calculation is included in Figure 196.

```
Four-Bit Register with Equality & Parity outputs

    ==== P18CV8 Programmed Logic ====

ODD      = (  !Q1 & Q2 & Q3 & Q4
           #   Q1 & !Q2 & Q3 & Q4
           #   Q1 & Q2 & !Q3 & Q4
           #   !Q1 & !Q2 & !Q3 & Q4
           #   Q1 & Q2 & Q3 & !Q4
           #   !Q1 & !Q2 & Q3 & !Q4
           #   !Q1 & Q2 & !Q3 & !Q4
           #   Q1 & !Q2 & !Q3 & !Q4 );

EQ     = !(  !D1 & Q1
          #   D1 & !Q1
          #   !D2 & Q2
          #   D2 & !Q2
          #   !D3 & Q3
          #   D3 & !Q3
          #   !D4 & Q4
          #   D4 & !Q4 );

Eout     = !(  EQ & !Ein );

Oout     = (  !ODD & Oin
             #   ODD & !Oin );

Q4.D   = (  D4 ); " ISTYPE 'BUFFER'
Q4.C   = (  !Clk );

Q3.D   = (  D3 ); " ISTYPE 'BUFFER'
Q3.C   = (  !Clk );

Q2.D   = (  D2 ); " ISTYPE 'BUFFER'
Q2.C   = (  !Clk );

Q1.D   = (  D1 ); " ISTYPE 'BUFFER'
Q1.C   = (  !Clk );
```

Figure 196. Four-Bit Register Summary Information.

```
==== P18CV8 Chip Diagram ====

                                P18CV8

                          +---------\       /---------+
                          |          \     /          |
                          |           -----           |
                     Clk  |  1                   20   |  Vcc
                          |                           |
                      D4  |  2                   19   |  Q4
                          |                           |
                      D3  |  3                   18   |  Q3
                          |                           |
                      D2  |  4                   17   |  Q2
                          |                           |
                      D1  |  5                   16   |  Q1
                          |                           |
                          |  6                   15   |  !EQ
                          |                           |
                          |  7                   14   |  ODD
                          |                           |
                     Ein  |  8                   13   |  !Eout
                          |                           |
                     Oin  |  9                   12   |  Oout
                          |                           |
                     GND  |  10                  11   |
                          |                           |
                          |                           |
                          `---------------------------'


==== P18CV8 Resource Allocations ====

      Device          | Resource  |   Design    |    Part     |
     Resources        | Available | Requirement | Utilization |  Unused
======================|===========|=============|=============|==============
                      |           |             |             |
Dedicated input pins  |     10    |       7     |      7      |     3 ( 30 %)
Combinatorial inputs  |     10    |       7     |      7      |     3 ( 30 %)
Registered inputs     |      -    |       0     |      -      |     -
                      |           |             |             |
Dedicated output pins |      -    |       2     |      -      |     -
Bidirectional pins    |      8    |       6     |      8      |     0 (  0 %)
Combinatorial outputs |      -    |       4     |      -      |     -
Registered outputs    |      -    |       4     |      -      |     -
Reg/Com outputs       |      8    |       -     |      8      |     0 (  0 %)
Two-input XOR         |      -    |       0     |      -      |     -
                      |           |             |             |
Buried nodes          |      -    |       0     |      -      |     -
Buried registers      |      -    |       0     |      -      |     -
Buried combinatorials |      -    |       0     |      -      |     -
```

Figure 196. Four-Bit Register Summary Information. (continued)

```
    ==== P18CV8 Product Terms Distribution ====

           Signal          |    Pin    | Terms | Terms | Terms
            Name           | Assigned  | Used  |  Max  | Unused
===============================|===========|=======|=======|=======
ODD                            |    14     |   8   |   8   |   0
EQ                             |    15     |   8   |   8   |   0
Eout                           |    13     |   1   |   8   |   7
Oout                           |    12     |   2   |   8   |   6
Q4.REG                         |    19     |   1   |   8   |   7
Q3.REG                         |    18     |   1   |   8   |   7
Q2.REG                         |    17     |   1   |   8   |   7
Q1.REG                         |    16     |   1   |   8   |   7


     ==== List of Inputs/Feedbacks ====

Signal Name                    | Pin       | Pin Type
===============================|===========|=========
D4                             |     2     | INPUT
D3                             |     3     | INPUT
D2                             |     4     | INPUT
D1                             |     5     | INPUT
Clk                            |     1     | CLK/IN
Q1                             |    16     | BIDIR
Q2                             |    17     | BIDIR
Q3                             |    18     | BIDIR
Q4                             |    19     | BIDIR
EQ                             |    15     | BIDIR
Ein                            |     8     | INPUT
ODD                            |    14     | BIDIR
Oin                            |     9     | INPUT


    ==== P18CV8 Unused Resources ====

 Pin    |  Pin    |   Product    | Flip-flop
Number  |  Type   |   Terms      |   Type
========|=========|==============|==========
    6   |  INPUT  |      -       |    -
    7   |  INPUT  |      -       |    -
   11   |  INPUT  |      -       |    -
```

Figure 196. Four-Bit Register Summary Information. (continued)

2. Control State Machine

Information regarding the control state machine is included in Figure 197.

```
Control State Machine

    ==== P18CV8 Programmed Logic ====

iStrb     = (  !Strb );

s3.D   = (  s3.FB & s1.FB
         #   s2.FB & !s1.FB & s0.FB
         #   s3.FB & !s0.FB ); " ISTYPE 'BUFFER'
s3.C   = (  Clk );

s2.D   = (  !s3.FB & s2.FB & !s1.FB
         #   s2.FB & !s1.FB & s0.FB
         #   s3.FB & s1.FB & !s0.FB
         #   !s3.FB & !s1.FB & !s0.FB & iStrb & !Ein ); " ISTYPE 'BUFFER'
s2.C   = (  Clk );

s1.D   = (  s3.FB & s2.FB & s1.FB
         #   s3.FB & !s2.FB & !s0.FB ); " ISTYPE 'BUFFER'
s1.C   = (  Clk );

s0.D   = (  s3.FB & s2.FB & s1.FB
         #   s3.FB & !s2.FB & s0.FB
         #   !s3.FB & s2.FB & !s1.FB & s0.FB
         #   !s3.FB & s2.FB & !s1.FB & !iStrb ); " ISTYPE 'BUFFER'
s0.C   = (  Clk );

out.D   = (  !s3.FB & !s1.FB & !s0.FB
          #   s3.FB & Din
          #   !s3.FB & !s2.FB & !s1.FB & !Par ); " ISTYPE 'BUFFER'
out.C   = (  Clk );

busy.D   = (  s3.FB
           #   s2.FB & !s1.FB
           #   !s1.FB & s0.FB ); " ISTYPE 'BUFFER'
busy.C   = (  Clk );

Ackn.D   = (  s3.FB
           #   !s2.FB & !s1.FB
           #   !s1.FB & s0.FB ); " ISTYPE 'BUFFER'
Ackn.C   = (  Clk );
```

Figure 197. Control State Machine Summary Information.

```
==== P18CV8 Chip Diagram ====

                                P18CV8

                          +---------\       /---------+
                          |          \     /          |
                          |           -----           |
                     Clk  |  1                   20   |  Vcc
                          |                           |
                     Din  |  2                   19   |  s0
                          |                           |
                          |  3                   18   |  s1
                          |                           |
                          |  4                   17   |  s2
                          |                           |
                          |  5                   16   |  s3
                          |                           |
                          |  6                   15   |  iStrb
                          |                           |
                          |  7                   14   |  out
                          |                           |
                     Ein  |  8                   13   |  Ackn
                          |                           |
                     Par  |  9                   12   |  busy
                          |                           |
                     GND  |  10                  11   |  Strb
                          |                           |
                          |                           |
                          `---------------------------'


==== P18CV8 Resource Allocations ====

      Device          | Resource  |   Design    |    Part     |
     Resources        | Available | Requirement | Utilization |  Unused
======================|===========|=============|=============|==============
                      |           |             |             |
Dedicated input pins  |     10    |       5     |      5      |     5 ( 50 %)
Combinatorial inputs  |     10    |       5     |      5      |     5 ( 50 %)
Registered inputs     |      -    |       0     |      -      |     -
                      |           |             |             |
Dedicated output pins |      -    |       7     |      -      |     -
Bidirectional pins    |      8    |       1     |      8      |     0 (  0 %)
Combinatorial outputs |      -    |       1     |      -      |     -
Registered outputs    |      -    |       7     |      -      |     -
Reg/Com outputs       |      8    |       -     |      8      |     0 (  0 %)
Two-input XOR         |      -    |       0     |      -      |     -
                      |           |             |             |
Buried nodes          |      -    |       0     |      -      |     -
Buried registers      |      -    |       0     |      -      |     -
Buried combinatorials |      -    |       0     |      -      |     -
```

Figure 197. Control State Machine Summary Information. (continued)

```
==== P18CV8 Product Terms Distribution ====

          Signal          |    Pin    | Terms | Terms | Terms
           Name           | Assigned  | Used  |  Max  | Unused
=========================|===========|=======|=======|=======
iStrb                    |    15     |   1   |   8   |   7
s3.REG                   |    16     |   3   |   8   |   5
s2.REG                   |    17     |   4   |   8   |   4
s1.REG                   |    18     |   2   |   8   |   6
s0.REG                   |    19     |   4   |   8   |   4
out.REG                  |    14     |   3   |   8   |   5
busy.REG                 |    12     |   3   |   8   |   5
Ackn.REG                 |    13     |   3   |   8   |   5


     ==== List of Inputs/Feedbacks ====

Signal Name                | Pin       | Pin Type
===========================|===========|=========
Clk                        |     1     | CLK/IN
Strb                       |    11     | INPUT
iStrb                      |    15     | BIDIR
Ein                        |     8     | INPUT
Din                        |     2     | INPUT
Par                        |     9     | INPUT


    ==== P18CV8 Unused Resources ====

 Pin   |  Pin   |   Product   | Flip-flop
Number |  Type  |   Terms     |   Type
=======|========|=============|==========
    3  | INPUT  |      -      |    -
    4  | INPUT  |      -      |    -
    5  | INPUT  |      -      |    -
    6  | INPUT  |      -      |    -
    7  | INPUT  |      -      |    -
```

Figure 197. Control State Machine Summary Information. (continued)

3. Eight-to-One Multiplexer

Information regarding the eight-to-one multiplexer is included in Figure 198.

```
Eight to One Multiplexer

    ==== P18CV8 Programmed Logic ====

out      = (  s0 & !s1 & !s2 & a0
           #   s0 & s1 & !s2 & a1
           #   s0 & s1 & s2 & a2
           #   !s0 & s1 & s2 & a3
           #   !s0 & s1 & !s2 & a4
           #   !s0 & !s1 & !s2 & a5
           #   !s0 & !s1 & s2 & a6
           #   s0 & !s1 & s2 & a7 );

nClk     = (  !Clk );


    ==== P18CV8 Chip Diagram ====

                                  P18CV8

                          +---------\       /---------+
                          |          \     /          |
                          |           -----           |
                          |  1                   20   |  Vcc
                          |                           |
                      a0  |  2                   19   |
                          |                           |
                      a1  |  3                   18   |
                          |                           |
                      a2  |  4                   17   |
                          |                           |
                      a3  |  5                   16   |  nClk
                          |                           |
                      a4  |  6                   15   |  out
                          |                           |
                      a5  |  7                   14   |  s2
                          |                           |
                      a6  |  8                   13   |  s1
                          |                           |
                      a7  |  9                   12   |  s0
                          |                           |
                     GND  |  10                  11   |  Clk
                          |                           |
                          |                           |
                          `---------------------------'
```

Figure 198. Eight-to-One Multiplexer Summary Information.

```
==== P18CV8 Resource Allocations ====

       Device          | Resource  |    Design     |     Part      |
      Resources        | Available | Requirement   | Utilization   | Unused
======================|===========|===============|===============|==============
                      |           |               |               |
Dedicated input pins  |     10    |       12      |       9       |     1 ( 10 %)
Combinatorial inputs  |     10    |        9      |       9       |     1 ( 10 %)
Registered inputs     |      -    |        0      |       -       |     -
                      |           |               |               |
Dedicated output pins |      -    |        2      |       -       |     -
Bidirectional pins    |      8    |        0      |       5       |     3 ( 37 %)
Combinatorial outputs |      -    |        2      |       -       |     -
Registered outputs    |      -    |        0      |       -       |     -
Reg/Com outputs       |      8    |        -      |       2       |     6 ( 75 %)
Two-input XOR         |      -    |        0      |       -       |     -
                      |           |               |               |
Buried nodes          |      -    |        0      |       -       |     -
Buried registers      |      -    |        0      |       -       |     -
Buried combinatorials |      -    |        0      |       -       |     -


    ==== P18CV8 Product Terms Distribution ====

           Signal            |    Pin    | Terms | Terms | Terms
            Name             | Assigned  | Used  |  Max  | Unused
=================================|==========|=======|=======|=======
out                              |   15     |   8   |   8   |   0
nClk                             |   16     |   1   |   8   |   7


     ==== List of Inputs/Feedbacks ====

Signal Name                      | Pin       | Pin Type
================================ |==========|=========
s0                               |   12      | BIDIR
s1                               |   13      | BIDIR
s2                               |   14      | BIDIR
a0                               |    2      | INPUT
a1                               |    3      | INPUT
a2                               |    4      | INPUT
a3                               |    5      | INPUT
a4                               |    6      | INPUT
a5                               |    7      | INPUT
a6                               |    8      | INPUT
a7                               |    9      | INPUT
Clk                              |   11      | INPUT


    ==== P18CV8 Unused Resources ====

 Pin    |  Pin    |    Product    | Flip-flop
Number  |  Type   |    Terms      |   Type
========|=========|===============|==========
   17   |  BIDIR  | NORMAL  8     |    D
   18   |  BIDIR  | NORMAL  8     |    D
   19   |  BIDIR  | NORMAL  8     |    D
```

Figure 198. Eight-to-One Multiplexer Summary Information. (continued)

APPENDIX G. COMMAND TRANSMISSION PROGRAM

The Parallel Port Data Modulator provides a means to convert command bytes from a computer parallel port to the required serial bit stream. A software interface is needed to place the bytes on the parallel port for the command modulator to read. C++ was used to write such an interface.

A. PARALLEL PORT COMMAND TRANSMISSION

The command transmission program accepts byte values from the user and places those bytes onto the parallel port. The program checks to determine if the peripheral is busy before placing the command on the port. If the port remains busy for an extended period, the program notifies the user. After placing the command on the parallel port, the transmission interface checks for modulator acknowledgement. Once acknowledgement is received or a wait period expires, the program prompts the user for the next command byte. The source code for the test program is provided in Figure 199.

```
//*******************************************************
// Program: parallel.cpp
// Name: Jeff Link
//
// Parallel Command Transmitter, ver 1.2
// Operating Environment: DOS
// Compiler: Borland C++ ver 5.02
// Date: 10 March 1999
//
// Description:  This program issues user entered command bytes to the
//     parallel port and waits for the command to be acknowledged.  The
//     program is a driver for the Parallel Port Data Modulator developed
//     in conjunction with the Tactor Interface Chip research project.
//*******************************************************

#include <iostream.h>;
#include <dos.h>;

int getValue();

void main(void) {
  cout << "Parallel Command Transmitter, ver 1.2" << endl;
  cout << "Jeff Link (c) 1999 All rights reserved.\n" << endl;
  int *portlist = (int *)0x408;
  int lpt1data = *portlist;
  int lpt1stat=lpt1data+1;
  int lpt1cont=lpt1data+2;
  cout << "LPT1 detected at " << lpt1data << endl;
  int val,ii,resp;
  while((val=getValue())<256 && val>-1) {
    ii=0;                                  // this loop waits if port is busy
    while ((resp=(inportb(lpt1stat)&0x80))==0 && ii <= 1000) {
      if (ii%100==0)
        cout << "Parallel port is busy for " << ii << " cycles." << endl;
      ++ii;
    }
    if (resp != 0) {
      outportb(lpt1data,val);              // put data on port
      outportb(lpt1cont,0x01);             // send strobe signal
      ii=0;                                // this loop waits for acknowledgement
      while ((resp=inportb(lpt1stat)&0x40)==0 && ii <= 1000) {
        if (ii%100==0)
          cout << "Waiting " << ii << " cycles for acknowledgement." << endl;
        ++ii;
      }
      outportb(lpt1cont,0x00);             // clear strobe signal
      if (resp == 0)
        cout << "No data acknowledgement received." << endl;
    }
    else
      cout << "Command transmission aborted; no data sent." << endl;
  }
}

int getValue() {
  int val;
  cout << "Enter value for BYTE to send (>255 quits): ";
  cin >> val;
  return val;
}
```

Figure 199. Command Transmission Driver C++ source code.

APPENDIX H. GOMAC CONFERENCE PAPER

The research presented in this thesis was also published and presented at the 1999 Government Microcircuit Applications Conference. The four-page article, Reference 2, is included as Figure 200, Figure 201, Figure 202, and Figure 203.

A BUS INTERFACE CHIP FOR TACTILE COMMUNICATIONS

Jeffrey P. Link and Douglas J. Fouts
U.S. Naval Postgraduate School
Monterey, CA 93943

ABSTRACT

Implementation of tactile communication requires rapid parametric data transfer along a common bus. The developed communication protocol and application-specific interface chip enable precise control of multiple tactors to convey information to military users.

INTRODUCTION

Touch is a physical sensory input not commonly associated with conveying computer information. Yet, when a person is touched, the response is immediate and often involuntary. The immediate nature of touch response makes it ideal for communicating critical information. Tactile communication can also be the most appropriate interface for specific types of information when existing visual and auditory activities cannot be compromised[1].

The Naval Aerospace Medical Research Laboratory built a rudimentary implementation of tactile communication in their Tactile Situation Awareness System (TSAS). To refine this interface, the Naval Postgraduate School developed a compact communication topology for connecting each tactile transmitter (tactor) to the controlling microprocessor. Serial communications were selected for this application to minimize the number of conductors required for data transfer.

An application-specific Tactor Interface Chip (TIC) provides the necessary hardware to realize the serial communication scheme. Each tactor in a forty-element array will include a TIC, as shown in Figure 1, that controls tactor activation. This hardware combination forms an "intelligent tactor" that shifts waveform creation from the microprocessor to the individual tactors. The resulting decrease in computational load allows use of a slower microprocessor, decreasing system power consumption.

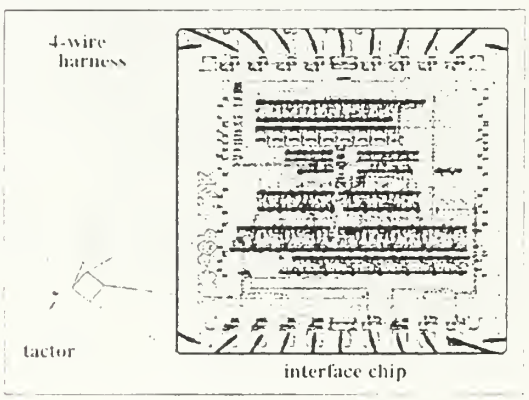


Figure 1. The Tactor Interface Chip (TIC) embedded in the casing of each tactile transmitter (tactor) controls application of power for waveform generation.

COMPETING DESIGN CONSTRAINTS

SIZE: Funding limits forced microchip size to be a primary constraint. Since component interconnections consume the majority of VLSI layout area[2], chip size primarily bounds the number of circuit components. This sharply limits circuit complexity and fundamentally affected design decisions.

POWER: Since the tactile interface is a stand-alone bridge between the information source and the human user, each TIC must draw minimum current from the battery-powered system. Using the smallest possible CMOS FETs throughout the circuit minimizes power consumption of the elementary components. Aggressively simplifying the logic structure further reduced power requirements.

SPEED: Small transistor size adversely influences response time. Minimum transistor size is sufficient at a 1 MHz clock speed unless long interconnects or several components must be driven. Individual elements were resized based on their output loading.

CONTROL STRUCTURE

ADDRESS: Transmission of tactile messages requires each tactor in the forty-element array to be capable of producing defined pulse shapes. These tactile signals can be independent or synchronized with several other tactors. Since the pulse shape parameters are transmitted on a common data bus, each TIC must be able to recognize commands meant to control the attached tactor. Unique identification is accomplished by assigning an "address" to each TIC. Use of a single TIC design for all tactors is possible by externally setting the address parameter by grounding TIC input pins. Planned modifications to this design are discussed in the "Future Improvements" section of this paper.

PULSE SHAPE: Tactors are repeatedly pulsed to convey information to the user. Changing the pulse duration and pulse rate creates different physical sensations and can be used to relate differing messages. Coordinated pulse shapes on adjacent tactors can produce an illusion of motion to relate additional information. Pulse shape production requires two parameters, pulse width and repetition period, illustrated in Figure 2. The TIC stores these values in data registers that are used to control tactor activation.

Figure 200. GOMAC Conference Paper (page 1 of 4).

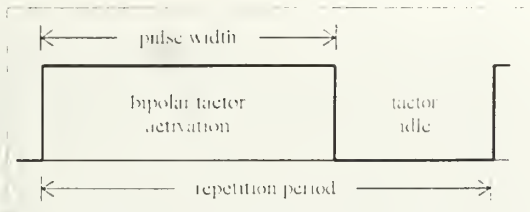


Figure 2. Tactor activation is controlled by the pulse width and repetition period values that are stored on the TIC.

COMMUNICATION PROTOCOL

An eight-bit communication scheme is utilized to ensure easy integration to different micro-controllers. The data words represent address, pulse width, and repetition period commands as summarized in Table 1. The Universal

Word Format	Meaning
0 X X X X X X X	7-bit Address
1 0 X X X X X X	6-bit Pulse Width
1 1 X X X X X X	6-bit Repetition Period

Table 1. Format of the three command types allows rapid address comparison and pulse-shape parameter storage.

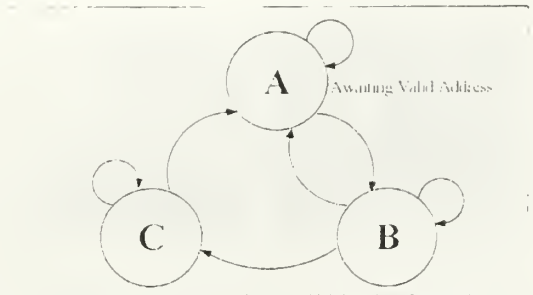
Synchronous/Asynchronous Receiver-Transmitter (USART) data format is used to package the command bytes into a serial bit stream that can be easily detected. The packet is illustrated in Figure 3 and includes a start bit, eight data bits, a parity bit and a stop bit. This data package format also provides basic fault protection. The data line remains at a logic "1" while idle.

Figure 3. Standard USART format provides a discernible package and basic error detection.

OPERATIONAL DESCRIPTION

The TIC continuously monitors the serial data bus and decodes the bit stream to detect and latch command bytes onto an internal command bus. When the bytes are latched, a data-valid signal triggers command evaluation and subsequent control of the TIC operational state. The state diagram in Figure 4 illustrates the TIC operating sequence.

Initially, the TIC is in a monitor state waiting to receive a valid address. When an appropriate address is received, the TIC shifts to a condition that waits for a command to set the register values. When a register command is received, the TIC enters a state that responds to all register commands until an address is detected, marking the end of the command cycle. This operating sequence provides easy



Initial State	Transition Condition	Next State
All	Reset asserted	A
A	Parity valid & valid address	B
B	Any other address received	B
B	Any register command received	C
C	Any address received	A
C	Any register command received	C

Figure 4. The Operating Sequence ensures that each TIC only responds to properly addresses commands.

control and allows the identical command to be sent to several tactors simultaneously.

If the stored pulse width is non-zero, the TIC activates the attached tactor in a pattern defined by the stored values of pulse width and repetition period. Any change to either waveform parameter will cause the TIC to reset the wave counter, synchronizing all tactors that simultaneously receive the command.

FUNCTIONAL COMPONENTS

TIC design focused on three areas: detecting and latching serial commands onto the command bus, interpreting commands to set the activation parameters, and generating bipolar current to drive the attached tactor. Each functional area was designed to operate independently with well-defined inputs and outputs. This modular approach was critical to the design and testing of lower-level components.

SERIAL DATA RECEIVER. The Serial Data Receiver (Figure 5) continuously monitors the input data line to detect and latch transmitted packets onto the command bus. It consists of a twelve-bit shift register, a validity checker, and an eight-bit latch. The most recent twelve data bits are stored in the shift register and compared to the USART format rules. When a string of bits is detected that meets the validity check, the command byte of the data packet is latched onto the command bus. The latch signal also triggers a "Bus Data Valid" signal that enables the command decoder. A feedback path partially clears the shift register to ensure that two immediately sequential data packets do not produce an erroneous command detection.

Figure 201. GOMAC Conference Paper (page 2 of 4).

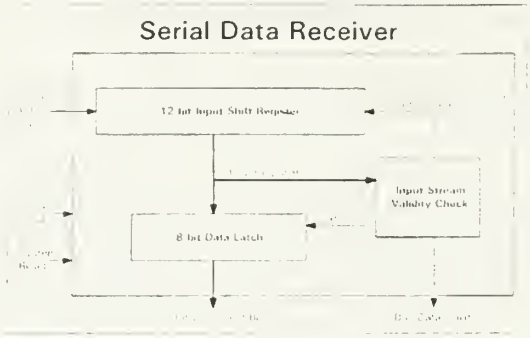


Figure 5. The Serial Data Receiver extracts the 8-bit commands from the serial command stream.

COMMAND DECODER AND CONTROLLER. The Command Decoder and Controller (Figure 6) evaluates the received commands and adjusts the internally stored waveform parameters if the command is properly addressed to the attached tactor. It consists of a sequence controller, address comparitor, and two six-bit registers. The sequence controller is a state machine (refer to Figure 4) that causes the TIC to react only to properly addressed commands. The address reference maintains a unique address for the individual tactor. The TIC ignores all received commands until the address comparitor detects its assigned address (or the "all call" address). It then updates the stored pulse width and repetition period with every new register command. Then, when an address is received, the TIC returns to a monitor condition and waits for the next properly addressed command.

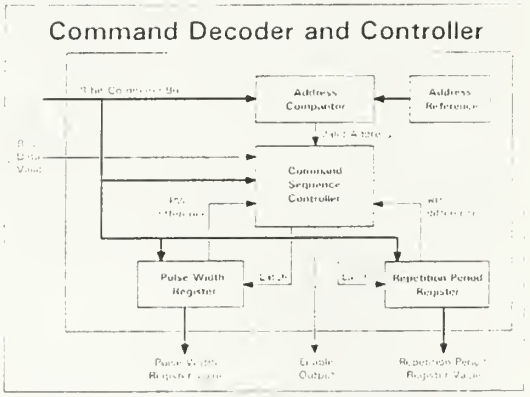


Figure 6. The Command Decoder and Controller interprets commands and updates register values as appropriate.

TACTOR POWER CONTROLLER. The Tactor Power Controller (Figure 7) converts the input data signals into pulsed bipolar power that is applied directly to the tactor. A frequency divider reduces the 1 MHz clock to a selectable tactor oscillating frequency and a 62.5 Hz down counter clock. The oscillator frequency is applied to the power oscillator to produce alternating current for the tactor. The power controller uses two synchronized down counters to create the stored wave shape by activating and disabling the power oscillator output. The control logic produces the wave cycle by clearing and loading both down counters based on the down counter conditions and the "enable output" signal.

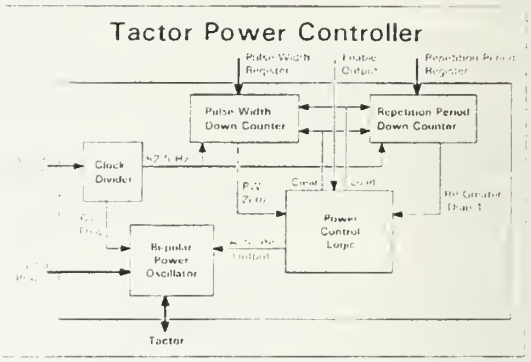


Figure 7. The Tactor Power Controller applies power to the tactor based on stored wave-shape parameters.

SPECIAL DESIGN FEATURES

Several features of the current design provide enhanced system performance. Some features are included primarily for chip testing and evaluation.

MULTIPLE COMMAND PACKET ADDRESSING. The operating-state transition definitions allow a command byte stream that includes multiple TIC addresses. This feature allows a command to activate several tactors with a single, synchronized wave-shape.

ALL-CALL ADDRESS. One address value is reserved to represent a valid address for all TICs. This feature is intended for use with a system reset command or when testing the entire communication array.

DUAL RESET CIRCUIT. The analog response of the circuit components is used to produce an initial reset signal for the first 200 nS of TIC operation. The reset ensures that all components establish a known condition when the circuit is started. A selectable, low-voltage reset is included to protect the system from an erratic response caused by low input voltage.

SELECTABLE OSCILLATOR FREQUENCY. An input jumper provides two tactor oscillation frequencies: 125 Hz and 250 Hz. This feature allows the TIC to be used with different tactors during prototype evaluation.

SELECTABLE ADDRESS. By including the TIC address as an external input, a single TIC design is used for all tactors in the communication array. In addition to enhancing prototype testing, this approach will be retained in future versions to ensure that a single "intelligent tactor" can function in every possible array position.

Figure 202. GOMAC Conference Paper (page 3 of 4).

FUTURE IMPROVEMENTS

PROGRAMMABLE ADDRESSES. Use of programmable-gates will allow the TIC address to be electronically assigned. Additionally, multiple address registers may be included to allow issuing TIC commands to groups of tactors simultaneously using a single address.

PROGRAMMABLE OSCILLATION FREQUENCY. Adding a frequency register would allow the TIC to vary the tactor activation frequency. This could be implemented either through an external jumper setting or as an additional command.

PROGRAMMABLE VOLTAGE SHAPING. Currently the tactor voltage is applied in a bipolar square wave. Tactor response may vary noticeably when a sine wave is used to drive the tactor. Use of a varying voltage would also reduce the switching transients created by the square-wave current spikes.

EXPANDED INSTRUCTION SET. Many additional instructions could be included in the basic TIC control language. This change requires restructuring the command protocol and making significant changes to the TIC design. Including a programmable micro-code register into the system would provide the most flexible solution. However, this approach is not a priority due to its huge increase in circuit complexity and required layout area.

TWO-WAY COMMUNICATIONS. A change to the fundamental system paradigm might incorporate the ability for real-time feedback to the controller. The status data could include all current TIC parameters. Incorporating an onboard vibration sensor could also provide actual indication of tactor operating parameters.

PROJECT STATUS

The TIC is completely designed and simulated using Cadence VLSI design software. Exhaustive simulation shows that the system operates precisely as designed. The circuit performed flawlessly at speeds up to 5 MHz.

The National Science Foundation VLSI design program facilitated TIC fabrication through the MOSIS[3] service. MOSIS provides low-cost prototyping and production service for VLSI circuit development.

Initial chip testing produced no detectable output. Visual examination of the chip showed areas of possible contamination during the fabrication process. Subsequent chip evaluation with a scanning electron microscope revealed contamination between power lines and between data paths (Figure 8). Figure 9 shows aluminum oxidation detected along some of the conductors. Further evaluation is in progress to precisely identify the faults in each chip.

SUMMARY

Tactile communication is an extremely viable method of conveying information without impeding other sensory inputs. In many applications, tactile messages may be most appropriate due to their intuitive and covert nature.

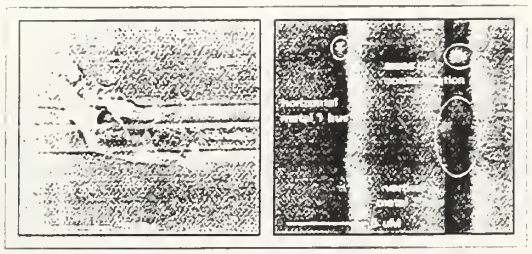


Figure 8. Scanning Electron Microscope images of possible power shorts (left) and command-bus shorts (right).

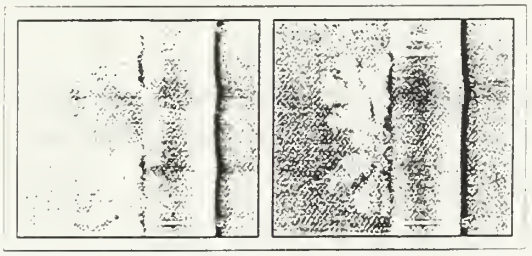


Figure 9. Scanning Electron Microscope images of areas with aluminum oxidation.

Previously, tactile communication has been experimental and limited, lacking methods to take the technology beyond the laboratory. The Naval Postgraduate School has developed a communication protocol and a tactor interface chip that will advance tactile communication beyond its current academic environment.

Implementation of this concept is currently awaiting VLSI fabrication. As more funding becomes available, many improvements are planned for the next generation of Tactor Interface Chips. The Naval Postgraduate School is anxious to advance this technology for military and public applications.

REFERENCES

[1] Hong Z. Tan and Alex Pentland, "Tactual Displays For Wearable Computing", *Proceedings of the First International Symposium on Wearable Computers*, IEEE, pp. 84-89, 1997.

[2] Neil H.E. Weste and Kamran Eshraghian, *Principles of CMOS VLSI Design*, Addison-Wesley, 1993.

[3] http://www.mosis.org/

Figure 203. GOMAC Conference Paper (page 4 of 4).

LIST OF REFERENCES

1. Jan Axelson, *Parallel Port Complete*, Lakeview Research, 1996.

2. Jeffrey P. Link and Douglas J. Fouts, "A Bus Interface Chip for Tactile Communications," *Digest of Papers for the 1999 Government Microcircuit Applications Conference*, pp. 460-463, March 1999.

3. Victor P. Nelson, H. Troy Nagle, Bill D. Carroll, and J. David Irwin, *Digital Logic Circuit Analysis and Design*, Prentice Hall, 1995.

4. H. Tan and A. Pentland, "Tactual Displays For Wearable Computing," *Proceedings of the First International Symposium on Wearable Computers*, IEEE, pp. 84-89, 1997.

5. David Pellerin and Michael Holley, *Digital Design using ABEL™*, PTR Prentice Hall, 1994.

6. Donald E. Thomas and Philip R. Moorby, *The Verilog® Hardware Description Language, 3rd edition*, Kluwer Academic, 1996.

7. Paul W. Tuinenga, *SPICE: A Guide to Circuit Simulation & Analysis Using PSpice®*, Prentice Hall, 1988.

8. John F. Wakerly, *Digital Design: Principles and Practices, 2nd edition*, Prentice Hall, 1994.

9. Neil H. E. Weste and Kamran Eshraghian, *Principles of CMOS VLSI Design: A Systems Perspecitve, 2nd edition*, Addison-Wesley, 1993.

10. M. Zlotnik, "Applying Electro-Tactile Display Technology to Fighter Aircraft -- Flying With Feeling Again," *Proceedings of the National Aerospace and Electronics Conference*, IEEE, pp. 191-197, 1988.

INITIAL DISTRIBUTION LIST

No. Copies

1. Defense Technical Information Center 2
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

3. RADM Robert C. Chaplin, USN 1
   Naval Postgraduate School
   1 University Circle
   Monterey, CA 93943

4. Chairman, Code EC 1
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA 93943-5121

5. Professor Douglas J. Fouts, Code EC/Fs 2
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA 93943-5121

6. Professor Jon T. Butler, Code EC/Bu 2
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, CA 93943-5121

7. Angus H. Rupert, M.D., Ph.D 1
   CDR, Navy Medical Corps
   Naval Aerospace Medical Research Laboratory
   51 Hovey Road
   NAS Pensacols, FL 32504

8. LCDR Jeffrey P. Link 4
   11808 Mallard Rd.
   Mason Neck, VA 22079-4111

DUDLEY KNOX LIBRARY
3 2768 00356916 1